SHARP

SHARP
COMPUTER
SOFTWARE

X68000用

C COMPILER PRO-68K ver2.0

Cリファレンスマニュアル

X68000用

C COMPILER PRO-68K ver2.0

Cリファレンスマニュアル

SHARP

はじめに

本書は、X68000上で動作する「C compiler PRO-68K ver2.0」(以下、XCコンパイラと表記します)の言語仕様についてまとめたものです。
なお、別冊の「Cユーザーズマニュアル」にXCコンパイラの商品構成、動作環境の作成方法、各マニュアルの内容が説明されています。
初めて本プログラムをご使用になる方は、必ず「Cユーザーズマニュアル」を先にご覧ください。

Cは、UNIXオペレーションシステムを記述するために開発された言語です。
そのため、今までアセンブラでなければ書けなかった、ハードウェアに密着した記述が可能になっています。
また、繰り返しや条件分岐などの制御構造と、豊富に用意されたライブラリを使って、わかりやすく信頼性の高いプログラムを作成できます。
XCコンパイラは、C言語の標準であるANSI規格準拠をさらに強化し、プロトタイプ宣言をデフォルトに変更した最新仕様のCコンパイラです。
他の処理系で動作しているプログラムを移植するときなどは、本マニュアルで仕様の互換性を確認してください。

本書では、実際のコンパイラの操作や、提供されているライブラリについては書かれていません。
コンパイラの操作については、「Cユーザーズマニュアル」を、ライブラリ関数については、「Cライブラリマニュアル VOL. 1」、「Cライブラリマニュアル VOL. 2」をそれぞれ参照してください。

※ UNIXはAT&Tが開発し、ライセンスしています。

本書の構成

本書は、次の５つの章と付録から構成されています。

第１章　Ｃプログラムの基本要素

１つの言語を覚えるためには、まず、その言語のもっとも基本となる部分を確実に理解しなければなりません。
この章では、プログラムを書くうえで使用する文字セットや基本データ型などの、Ｃ言語の基本的な構成要素について説明しています。

第２章　式と演算子

Ｃ言語には、他の言語に比べて多くの演算子があります。
演算子の効果的な使いかたをマスターすれば、複雑な処理もすっきりしたアルゴリズムで記述できるようになります。
ここでは、式の書き表しかたと演算子の動作について説明します。
また、演算の結果おこるデータ型の変換についても説明しています。

第３章　制御構造と関数

効率のよい、保守のしやすいプログラムを書くためには、分岐や繰り返しなどの制御構造を十分理解することが重要です。
この章の前半では、制御構造を記述する文について説明します。
後半は、Ｃ言語で処理を実行する単位となる関数について説明してあります。

第４章　構造をもった型

Ｃ言語では、基本データ型と組み合わせて新しい型を定義することが可能です。
ここでは、配列、構造体、共用体という構造をもった型について説明します。
さらに、配列と密接な関係にあるポインタの概念についても、この章で説明しています。

第５章　プリプロセッサ

強力なテキスト処理を実行するプリプロセッサの存在が、システム記述言語という開発当初の枠を超えてＣ言語が利用されている理由の１つでしょう。
ここでは、プリプロセッサを使って実行できる、マクロ定義、他のソースプログラムのとり込み、条件つきコンパイルなどについて説明しています。

付　録　構文の要約

XC コンパイラの構文規則を要約してあります。
リファレンスとして利用してください。

*　　*　　*　　*　　*

各章の解説には、プログラムを書くうえで参考になるように、できるかぎり具体的な例題をつけてあります。
例題の中には、そのまま実行できるものも含まれていますので、実際にコンパイルして動作を確認できます。

CONTENTS

第1章　Cプログラムの基本要素

第2章　式と演算子

第 3 章　制御構造と関数

第 4 章　構造をもった型

第5章　プリプロセッサ

付　録　構文の要約

索　引

第１章

Ｃプログラムの基本要素

使用する文字セット

エスケープシーケンス

基本データ型と変数

本章では、C言語の基本要素について説明します。

Cプログラムは、英数字、記号、およびその組み合わせである予約語、そして演算子で記述されます。
プログラム全体は、関数と呼ばれるいくつかのサブルーチンで構成されます。
関数と演算子については、後の章で詳しく説明します。
また、Cにはいくつかの基本的なデータ型があります。
変数はすべて型をもって宣言されます。

1.1 使用する文字セット

Ｃのプログラムで使用する文字セットは、コンパイラに対して意味をもつ文字、数字、いくつかの記号の組み合わせから構成されています。
コンソールから入力される文字にはいろいろなものがありますが、Ｃの文字セットは、表示できる文字セットの一部分です。
ただし、文字列定数、文字定数、注釈の中にはすべての文字（たとえば、漢字やカナなど）が使用できます。
Ｃの文字セット以外の文字や誤った使いかたがされていると、コンパイラはエラーメッセージを出力します。

1.1.1 文字と数字

Ｃの文字セットには、英大文字、英小文字、10 進数字、記号（特殊文字）が含まれています。
記号は、単独または組み合わされて、いろいろな目的に使います。
プログラムを記述する文字を以下に示します。

英大文字
ABCDEFGHIJKLMNOPQRSTUVWXYZ

英小文字
abcdefghijklmnopqrstuvwxyz

10 進数字
0123456789

記号
! ” # % & ’ ( ) * + , − . ／ : ; < = > ? [
¥] ^ _ { | } ‾

これらの文字、数字、記号を使って、定数、識別子、予約語、演算子などを作ることができます。
コンパイラは、英大文字と英小文字を区別します。

たとえば、Thing と thing は同じ意味では使えません。
これ以外の文字（漢字やカナなど）は、文字定数、文字列定数、注釈の中のみで使用できます。

1.1.2 空白文字

スペース、タブ、改行、復帰、改ページ、復帰改行文字は、空白文字と呼びます。
空白文字は、プログラム内で定数や識別子などの定義項目とその他の項目を分けるために使います。
空白文字はいくつでも続けて書けるので、プログラムを読みやすくするために使うこともできます。
また、テキストファイルの終わりを表す文字として、ctrl-Z（16 進数で 1A）を使います。
コンパイラは ctrl-Z があると、それ以後のテキストを読み込みません。

1.1.3 エスケープシーケンス

エスケープシーケンスは、表示されない空白文字などを文字列定数の一部や文字定数として表現したものです。
エスケープシーケンスは、円記号（¥）と 1 個の文字か数字の組み合わせで表します。
C 言語で使うエスケープシーケンスを以下に示します。

¥a	警告ベルを鳴らす
¥b	バックスペース
¥f	改ページ
¥n	復帰改行
¥r	復帰
¥t	水平タブ
¥v	垂直タブ
¥'	シングルクォーテーション
¥"	ダブルクォーテーション
¥?	疑問符
¥¥	円記号
¥0	空文字

¥nnn ¥xnn または¥Xnn	8進数で表されたASCII文字 16進数で表されたASCII文字

上記以外の文字を円記号(¥)と組み合わせた場合、円記号は無視されます。
たとえば、'¥z'は'z'と同じになります。
また、すべてのASCII文字は、¥nnn（nnnは3桁の8進数）または¥xnn（nnは2桁の16進数）で表すことができます。
たとえば、水平タブは'¥011'または'¥x09'と表記できます。
16進数の最初の桁が0のときは省略できますが、少なくとも数字1桁は書かなければなりません。
先にあげた水平タブは、'¥x9'と表せます。
ただし、文字列定数の中では、すべての桁を指定するようにおすすめします。

```
"File¥x9not¥x9found"
```

たとえば、上の文字列の場合、2番目のエスケープシーケンスは¥x9fという16進数コードに解釈されるおそれがあります。

円記号（¥）は、文字列を継続させる文字としても使います。

例1

```
"abcdefghijk¥
lmnopqrstuvwxyz"
```

例2

```
"abcdefghijk¥
␣␣␣␣␣␣ lmnopqrstuvwxyz"
```

円記号（¥）のうしろの改行は無視されますが、継続する文字列の先頭にある空白文字は、他の文字と同じ扱いになります。
例2の文字列は、kとlの間に6個の空白（説明上、␣で表している）があります。

1.2 トークン

C 言語で記述されたソースプログラムは、コンパイル時にトークンと呼ばれる文字の集合に分けられます。
トークンは、予約語、識別子、演算子、定数、文字列リテラル、その他の区切子の6つからなります。
トークンは、ブランク、水平及び垂直のタブ、改行、改頁、注釈（注釈はトークンではない）といった空白文字、または他のトークンにより区切られます。
なお、空白文字はトークンを区切る場合以外は無視されます。

```
int a;
```

上の例では、空白文字で“int”、“a”の2つのトークンに分けられています。
また、コンパイラは、文字列をできるだけ長いトークンに分けようとします。

```
i+++j
```

たとえば上のような式の場合、コンパイラはi++をトークンの単位として扱います。
つまり(i++)+jと同じです。
トークンの区切りのあいまいさをなくすためには、カッコを使います。

```
i+(++j)
```

これで、コンパイラは++jをトークンとして理解できるようになります。

1.2.1 予約語

予約語は、コンパイラに対してあらかじめ定義された識別子です。
予約語はすべて英小文字で定義されており、定義と違う目的には使えません。
プログラム中の変数や関数を表す識別子は、予約語と重複しないように定義します。

次に予約語を示します。

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

予約語の再定義はできませんが、プリプロセッサを使って、他の識別子に置き換えることはできます。

プログラム例

```
#define unchar  unsigned char
           .
           .
           .
unchar  c;
```

詳しくは、「5．1　マクロ定義」を参照してください。

1.2.2 識別子

識別子は、プログラム内で使用する変数、関数、ラベルにユーザーが定義する名前で、たとえば、変数名や関数名などがあります。
識別子には、英大小文字、数字、アンダースコア（_）が使えますが、最初の文字は、英大小文字かアンダースコアでなくてはなりません。
コンパイラは、識別子の最初の 31 文字だけを認識します。
ただし、アンダースコアで始まる識別子は、あらかじめ定義してある識別子（標準ライブラリで使用）と重複することがありますので、なるべく使用しないでください。

次に正しい識別子の例を示します。

a
subl
power_2

コンパイラは、識別子の中の英大文字と英小文字を区別します。
次の識別子は、それぞれ別の識別子になります。

TOKEN
Token
token

1.2.3 演算子

演算子は、値の変化と代入を指定する文字の組み合わせです。
コンパイラは、この演算子をトークンとして解釈します。
次に演算子を示します。

```
!  ￣  +  －  *  ／  %  <<  >>  <=  <  >
>=  ==  !=  &  |  ^  &&  ||  ,  ?  :  ++  ――  =
+=  －=  *=  ／=  %=  >>=  <<=  &=  |=  ^=
－>  .  (  )  [  ]  sizeof  キャスト
```

複数の文字で構成される演算子の間には、空白文字を入れることはできません。
詳しくは、「2.2　演算子」を参照してください。

1.2.4 注　釈

注釈は、コンパイラには空白文字として扱われ、空白文字が書けるところにはどこにでも書けます。
注釈の書式は次の通りです。

／＊　「／＊」を含んでいない文字列中　＊／

文字列には改行文字を含めることができるので、注釈は２行以上にまたがってもかまいません。
ただし、ネストはできません。
また、コンパイラは注釈の文字列を無視しますので、注釈の中に予約語があってもエラーにはなりません。
なお、文字列、あるいは文字列リテラルの中には入れられません。

例１

```
／＊ Comments ＊／
```

例２

```
／＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊
  Comments
 ＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊／
```

例３

```
／＊ Comments ＊／
／＊  ERROR cannot／＊ nest ＊／comment ＊／
```

注釈の中に注釈は入れられないので、例３の下段はエラーになります。

1.3 基本データ型と変数

C言語には、いくつかの基本的なデータ型が用意されています。
基本データ型は、整数型、浮動小数点型、その他の型に分けることができます。
また、基本データ型から派生して作る型もありますが、派生型については「第4章　構造をもった型」でまとめて説明します。

1.3.1 基本データ型

基本データ型には、次の表に示すものがあります（カッコ内は省略形）。
これらの予約語は型指定子といい、変数の型を決定します。

整　数　型	浮動小数点	その他
char	float	enum
int	double	void
short int (short)	long double	
long int (long)		
unsigned char		
unsigned int		
unsigend short int (unsigned short)		
unsigned long int (unsigned long)		
signed char		
signed int (signed)		
signed short int (signed short)		
signed long int (signed long)		

整数型

整数型は、符号を持つものと持たないものに分けることができます。
符号を持たない型は、型指定子の前にunsignedをつけて表します。
また、符号を持つ型は、型指定子の前にsigned修飾子をつけて明示することができます。

char 型は、文字型ともいいますが、内部表現では８ビット符号つき整数になりますので、整数型に含まれています。
int 型は、コンパイラがもっとも扱いやすいサイズになりますので、他のコンパイラにプログラムを移植するときには注意します。
XC コンパイラでは、int 型は long 型と同じサイズになります。
次の表に、それぞれの整数型の記憶領域サイズと表現できる値の範囲を示します。

型	サイズ	表現できる値の範囲
char	8 ビット	−128〜127
int	32 ビット	−2, 147, 483, 648〜2, 147, 483, 647
short int	16 ビット	−32, 768〜32, 767
long int	32 ビット	−2, 147, 483, 648〜2, 147, 483, 647
unsigned char	8 ビット	0〜255
unsigned int	32 ビット	0〜4, 294, 967, 295
unsigned short int	16 ビット	0〜65, 535
unsigned long int	32 ビット	0〜4, 294, 967, 295
signed char	8 ビット	−128〜127
signed int	32 ビット	−2, 417, 483, 648〜2, 147, 483, 647
signed short int	16 ビット	−32, 768〜32765
signed long int	32 ビット	−2, 417, 483, 648〜2, 147, 483, 647

型指定子 int は、宣言のときにはすべて省略できます。

浮動小数点型

浮動小数点型には、サイズの違う float 型と double 型、long double 型の３種類があります。
次の表に、浮動少数点型のサイズと値の範囲を示します。
値の精度は、float 型で最大７桁、double 型、long double 型で最大 15 桁です。

型	サイズ	表現できる値の範囲
float	32 ビット	10^{-37}〜10^{38}
double	64 ビット	約 10^{-307}〜10^{308}
long double	64 ビット	約 10^{-307}〜10^{308}

その他の型

その他の型には、enum 型と void 型があります。
enum 型は列挙型ともいいます。
詳しくは、「1. 3. 3.　変数の宣言」を参照してください。
void 型は特別な型で、関数の宣言だけに使います。
void 型で宣言した関数は、値を返しません。

1.3.2 定数表現

定数は、プログラムの実行で変化しない値です。
C 言語の定数には、整数定数、浮動小数点定数、文字定数があります。

整数定数

整数定数は、整数値を表します。
10 進数、8 進数、16 進数の表現が使えます。
10 進数定数は、0 から 9 までの数字で表現します。
ただし、2 桁以上の数字では最初の桁は 1 から 9 までの数字で、0 は使えません。
負の数を表現するには、単項マイナス演算子（−）を定数の前に置きます。
8 進定数は、0 から 7 までの数字で表現します。
桁数に関係なく最初の数字は、必ず 0 でなくてはなりません。
16 進定数は、必ず 0x または 0X から始まります。
16 進数字として使える文字は、0 から 9 までの数字、a から f までの英小文字、A から F までの英大文字です。

8 進定数と 16 進定数は、符号なし整数として扱いますので、負の数は表現できません。
また、定数の末尾に u または U を付加すると unsigned 型、l または L を付加すると long 型に修飾されます。
ul または UL を付加すると unsigned long 型になります。
上記のような接尾子が付いていない場合、10 進数の定数の型は、その定数を表わすことのできる次の型の最初のものとなります。

int, unsigned int, long int, unsigned long int

1.3　基本データ型と変数

また、接尾子が付いていない 8 進数または 16 進数の定数の型は、その定数を表わすことのできる次の型の最初のものとなります。

int, unsigned int, long int, unsigned long int

10進数	8進数	16進数
12	014	0xc
255	0377	0xFF
36139	0106453	0x8D2B

例は左から、10 進、8 進、16 進のそれぞれの定数です。
同じ行には、同じ大きさの値を並べてあります。

浮動小数点定数

浮動小数点定数は、10 進数で符号つき実数を表します。
小数点を使った表現と合わせて、指数表現も使うことができます。
負の数は、単項マイナス演算子（－）を定数の前に置いて表します。
小数点の前の 0 は省略できます。
また、指数があるときにかぎり、小数点を含めた小数部分を省略できます。

例 1

```
34.56
3456e-2
3.456E1
```

例 2

```
0.005
.005
5e-3
```

例1、例2は、それぞれ同じ値を別の方法で表現したものです。
浮動小数点定数は、その末尾にfまたはFを付加するとfloat型、lまたはLを付加するとlong double型になります。
それ以外は、double型になります。

文字定数

文字定数は、文字、数字、記号をシングルクォーテーション（'）で囲ったものです。
文字にはエスケープシーケンスも含まれます。
囲む文字の数は1つだけです。
また、シングルクォーテーションと円記号は、必ずエスケープシーケンスで表します。

```
'A'
':'
'¥t'
'¥x16'
'¥'
'¥¥'
```

1.3.3 文字列定数

文字列定数は、文字例リテラルとも呼ばれ、0個以上の文字、数字、記号をダブルクォーテーション（"）で囲ったものです。
文字にはエスケープシーケンスも含まれます。
ダブルクォーテーションと円記号は、必ずエスケープシーケンスで表します。
文字列は、改行の前に円記号（¥）を置けば、次の行に継続できます。

例1

```
"string"
```

例 2

```
"Hit any key ¥n then continue"
```

例 3

```
"¥" MASTER FILE¥" not found ¥
program abort"
```

文字列定数は、記憶クラスが static で、文字配列の型を持ちます。
配列については、「4. 1　配列」を参照してください。

プログラム中で文字列定数を変更することはできません。
また、隣接している文字列定数は連結されます。
文字列の最後には、必ずヌル文字（'¥0'）が付加されます。
コンパイラは、ヌル文字によって文字列の終わりを認識します。

1.3.4 enum 型定数

enum 型は、整数定数のリストからなるデータ型です。
enum 型の変数には、リスト中の定数しか代入できません。
enum 型の変数は、次の書式で宣言します。
｛と｝は書式の一部です。

enum タグ名｛識別子リスト｝　識別子；

最後の識別子は省略でき、その形式を特に enum 型の型宣言といいます。型宣言によって、タグ名を型名に持つ enum 型のセットを定義します。
識別子リストは、カンマで区切って並べます。
それぞれの識別子は、int 型の定数になります。
したがって、enum 型変数は、int 型として扱います。
リスト中の識別子は、デフォルト値として０から始まり、宣言した順序に従い１ずつ増えた値をもっていますが、特定の値を指定することもできます。

例 1

```
enum month {
        err,            /*  ==  0  */
        Jan,            /*  ==  1  */
        Feb,            /*  ==  2  */
        Mar,            /*  ==  3  */
        Apr,            /*  ==  4  */
        May,            /*  ==  5  */
        Jun,            /*  ==  6  */
        Jul,            /*  ==  7  */
        Aug,            /*  ==  8  */
        Sep,            /*  ==  9  */
        Oct,            /*  == 10  */
        Nov,            /*  == 11  */
        Dec,            /*  == 12  */
        } year;
```

例 2

```
enum month  newyear;
```

例 3

```
int x = 15;
long endaddr = 0xFFFFFFL;
```

例 1 は、タグ名 month の enum 型変数 year を宣言しています。
year に代入できる値は、err から Dec までの定数です。
例 2 の month は、この宣言よりも前で定数か宣言があるものです。
タグ名 month を使って、変数 newyear を宣言しています。
例 3 は、enum 型の初期値の与えかたです。
識別子リストの中の Black から Red まではデフォルトの値で初期化され、
White は Yellow より 1 増えた値 7 で初期化されます。

1.3.5 変数の宣言・初期化

C言語では、変数を使う前に必ずその変数の名前と型を宣言します。
ここでは、基本データ型の変数宣言について説明します。
すべての基本データ型は、下の書式に従って宣言します。
［と］の中は、省略できることを表します。

［記憶クラス指定子］型指定子　識別子［, 識別子］……；

記憶クラス指定は省略できます。
記憶クラス指定子については、「1. 3. 6　記憶クラス」を参照してください。
識別子は、宣言する変数の名前です。
同じ型の変数であれば、識別子をカンマで区切って指定できます。
宣言するレベルが同じときには、必ず異なる変数名を指定します。
宣言のレベルについては、「1. 3. 8　変数の有効範囲」で説明します。
基本データ型の変数は、宣言するときに特定の定数式で初期値を持たせることができます。
これを、変数の初期化といいます。
初期化には記憶クラスと関係した制限がありますが、詳しくは「1. 3. 6　記憶クラス」を参照してください。

変数の初期化は、次の書式で行います。

識別子＝定数式

初期値を与える変数（識別子）の後のイコール記号（＝）に続けて、定数式を書きます。

例 1

```
int x;
```

例 2

```
char letter, number, but [20];
```

例 3

```
int x = 15;
long endaddr = 0xFFFFFFL;
```

例 1 は、int 型の変数 x の宣言です。
例 2 は、char 型の変数をカンマで区切って、1 度に宣言しています。
これは、次のように宣言したものと同じです。

```
char letter;
char number;
char buf [20];
```

例 3 は、整数定数で初期化しています。

1.3.6 記憶クラス

変数名は、ある値を入れておくメモリ上の領域につけられた名前です。
変数のための領域を確保する方法を指定するのが記憶クラス指定子です。
記憶クラス指定子には、次のものがあります。

auto, register, static, extern, typedef

次に、それぞれの記憶クラス指定子について説明します。
変数宣言のレベルや有効範囲に関しては、「1. 3. 8　変数の有効範囲」でもう1度まとめてあります。

auto 記憶クラス指定子

auto は、内部レベルの変数宣言にだけ指定できます。
内部レベルで記憶クラス指定子を省略した変数は、すべて auto 変数として扱います。
auto 変数は、メモリ上に動的に割りつけられ、宣言したブロック内だけで有効です。
プログラムの制御がブロックから他へ移ると、auto 変数は自動的に消滅します。
宣言のときに初期値を与えれば、割りつけられるたびにその値で初期化されます。
しかし、初期値を与えなければ値は不定になります。

register 記憶クラス指定子

register は、auto と同じく内部レベルの変数宣言にだけ指定できます。
register 変数は、レジスタ内に記憶領域を確保しますが、空レジスタがないときは auto 変数と同じ扱いになります。
宣言したブロック内だけで有効です。
自動的に初期化はされません。
したがって、初期値を与えない宣言では、値は不定となります。
また、register 宣言された変数は、実際には auto 変数と同じ扱いをされていても、そのアドレスを単項演算子&などで参照するのは文法違反となります。

static 記憶クラス指定子

static は、どのレベルの変数宣言でも指定できます。
static 変数は、宣言したブロック内だけで有効です。
メモリ上に静的に割りつけられるので、値はプログラムの終了まで保持されます。
プログラムの制御が、static 変数の宣言されているブロックに移ったときに1度だけ初期化されます（コンパイル時に初期化される）。
初期値を省略したときは、自動的に0で初期化されます。

extern 記憶クラス指定子

extern は、どのレベルの変数宣言でも指定できます。
外部レベルで記憶クラス指定子を省略した変数は、すべて extern 変数として扱います。
extern 変数は、プログラムのすべての範囲で有効です。
メモリ上に静的に割りつけられるので、値はプログラムの終了まで保持されます。
コンパイルのときに1度だけ初期化されます。
初期値を省略すると、自動的に0に初期化されます。
内部レベルでの extern 指定は、変数の参照宣言となります。
そのため、メモリへの割りつけは行われず、また初期値の指定もできません。

typedef 記憶クラスの指定子

typedef は他の記憶クラス指定子と違い、新しい型名を定義します。
メモリへの割りつけは行われません。

例1

```
typedef unsigned char BYTE;
BYTE x, y, z;
```

例2

```
typedef struct person {
        char name[20];
        int  age;
        int  sex;
        } CLASS;

CLASS my_class;
```

例3

```
typedef OPE(int, int);
OPE     add, sub, mult, div;
```

1.3.7 型修飾子

型修飾子は、int 型や char 型の型指定子の前について、その変数の型の特別な性質を表します。
型修飾子は、単体では使用しません。
型修飾子には、以下のものがあります。

const, volatile

なお、volatile は現在使用できませんが、予約されています。

const 型修飾子

const は、読み込み専用の値であることを表す修飾子です。
初期化はできますが、それ以後の書き込みは禁止される属性を持ちます。
もし、プログラム中で const 宣言した変数を書き換えるような操作を記述していると、コンパイル時に警告メッセージが表示されます。

1.3.8 変数の有効範囲

Cのプログラムは、いくつかのネストしたブロックに分けられます。
すべてのブロックの外側の部分を外部レベルといい、ブロックの内側は内部レベルといいます。
変数は、宣言されたレベルと記憶クラス指定子により有効範囲が決まります。
変数の有効範囲は、その変数名の通用範囲（スコープともいいます）と変数値の寿命という2つの概念で説明されます。
通用範囲は、プログラム内実行中に変数名で参照できる領域を指します。
また、寿命は変数値が保持される期間です。
変数は、寿命によりグローバル変数とローカル変数に分けることができます。
グローバル変数は、プログラムの実行終了までその値が保持され、ローカル変数は、制御の移動によって値が消滅します。
外部レベルで宣言された変数は、すべてグローバル変数となり、値はプログラムの終了まで保持されます。
ただし、通用範囲は、記憶クラス指定子によって変わります。
extern 変数は、宣言後すべての範囲で参照できます。
他のソースファイルからの参照も可能です。
指定子を省略した変数は、extern 変数として扱います。
static 変数の通用範囲は、宣言を含むソースファイル内だけです。
内部レベルで宣言された変数では、auto または register 変数はローカル変数になります。
extern はグローバル変数、static はローカル変数となります。
どの記憶クラスの変数も、通用範囲はその宣言を含むブロック内だけです。

次に例にあげたプログラムは、変数の有効範囲を示したものです。
関数 printf は、引数を出力する関数で、XC コンパイラの標準ライブラリ関数です（詳しくは、「C ライブラリマニュアル」を参照してください）。

1.3　基本データ型と変数

プログラム例

```
#include <stdio.h>

/*  ブロックレベル0   ( external )   */

int i = 0;

main()
{         /*  ブロックレベル１   (メインプログラム)   */

          i++;
          sub(i);

          {         /*  ブロックレベル２   */

                    auto int i = 0;

                    i++;
                    sub(i);

                    {         /*  ブロックレベル３   */
                              static int i = 5;

                              i--;
                              sub(i);
                    }         /*  ブロックレベル２へ戻る   */

                    i++;
                    sub(i);
          }         /*  ブロックレベル１へ戻る   */

          i--;
          sub(i);
}         /*  ブロックレベル０へ戻る   */

sub(x)
int x;
{         /*        ブロックレベル１   (サブプログラム)   */

          int i = 0;
          printf("%d %d¥n", x, i);
}
```

プログラムは、main と sub の２つの関数から構成されています。

グローバル変数は、外部レベルで宣言されている i と関数 main のブロックレベル３で static 変数で宣言されている i の２つです。

また、外部レベルで宣言された変数 i を除いて、通用範囲はその宣言を含むブロック内だけです。

関数は、それ自体がブロックになっているので、main のブロックレベル１と sub のブロックレベル１は、異なった内部レベルになります。

第 2 章

式と演算子

式の記述

演算子

演算子の優先順位

型変換

本章では、C言語の式の記述方法と演算子について説明します。

式は、オペランドと演算子の組み合わせです。
Cでは、式が値をもちます。
そのため、変数への代入も式と考えます。
式の中の各オペランドも、1つの値を表すので式になります。
また、Cは多くの演算子をもっています。
演算子は、式の処理の方法を指定します。
演算子には、優先度と評価順序があり、それによって、式が評価されます。
優先度は、カッコを使えばかえられます。
また、式によっては評価順序による副作用をおこすことがあります。
副作用がある式では、1つのオペランドの評価がその式の他の値に影響します。
式の値には型があります。
型は、代入などによって他の型に変換される場合がありますし、また、強制的に変換することもできます。

2.1 式の記述

Cの式は、オペランドと演算子を組み合わせて記述します。
オペランドは、定数、識別子、文字列、添字式、メンバ参照などです。
式そのものもオペランドとなります。
定数値をもつオペランドを定数式と呼びます。
オペランドにはすべて型があります。
また、オペランドは、型キャスト演算によって他の型にキャスト（はめこむ、変換する）することができます。

2.1.1 定数式

定数式とは、結果が定数になる式すべてを指します。
定数式のオペランドとして使えるものは、整数定数、文字定数、浮動小数点定数、enum（列挙）型定数、整数型や浮動小数点型への型キャスト式、他の定数式です。

例 1

```
123＋456
```

例 2

```
30.5 ＊ 2／(LIMIT ＋ 2)
```

例 3

```
'a'-'A'
```

例 4

```
10 + (x = 300)    /* error */
```

例 1 から例 3 は、正しい定数式です。
例 2 の LIMIT は、プリプロセッサ命令# define で定数として定義してあるものです。

2.1.2 添字式

添字式は、配列の要素やポインタを参照するための式です。

書 式

式 1 [式 2]

式 1 から式 2 で表される位置だけ先のアドレスにある値を表します。
式 1 はポインタ値で、式 2 は整数値です。
式 2 は、必ずカッコ（[ ]）で囲みます。
添字式の評価結果は、ポインタ値に整数値を加えたものを間接演算した値です。
一次元配列では、下の式の評価は同じになります。
ここでは、a はポインタ、b は整数です。

例

```
a [b]
* (a + b)
```

添字式では、二次元以上の配列は次のように記述します。

```
array [a] [b] [c]
```

これは、三次元の配列の要素を参照する式です。

2.1.3 メンバ参照式

メンバ参照式は、構造体や共用体を構成するデータであるメンバを参照するためのものです。
メンバ参照式は、参照するメンバの値と型をもちます。
メンバ参照式は、次の書式で記述します。

書　式

式．識別子
式－＞識別子

2つ目の書式は、構造体や共用体へのポインタです。
下の式の評価結果は、同じものになります。

プログラム例

```
(*pntr).name
pntr->name
```

2.1.4 型キャスト式

型キャストは、強制的な型変換を行います。
型キャスト式は、次の書式で記述します。

書　式

（型名）オペランド

型キャストの変換については、「2.4　型変換」で説明します。

プログラム例

```
(double) n
```

この例では、変数nの型をdouble型に変換します。

2.1.5 カッコについて

オペランドは、カッコで囲むことができます。
カッコは、演算子の優先順位をかえますが、式の型や値には影響しません。

(30＋6)／6

上の式では、30＋6 を評価した値が除算演算子（／）の左オペランドとなります。
結果は 6 です。
カッコがなければ、6／6 が先に評価されるので、結果は 31 になります。

2.2 演算子

Cの演算には、単項演算子、二項演算子、三項演算子があります。

単項演算子

単項演算子は、オペランドの前に置き、右から左に評価します。
単項演算子は、次の通りです。

!	NOT 演算子
~	1の補数演算子（チルド：マイナス記号ではない）
+	単項プラス演算子
−	単項マイナス演算子
++	インクリメント演算子
−−	デクリメント演算子
*	間接演算子
&	アドレス演算子
(型名)	キャスト演算子
sizeof	sizeof の演算子

インクリメント／デクリメント演算子だけは、オペランドの後にも置けます。

二項演算子

二項演算子は、左から右に評価します。
二項演算子は、次の通りです。

*　／　%　+　−	算術演算子
<<　>>	シフト演算子
<　>　<=　>=　==　!=	関係演算子
&　|	ビット演算子
&&　||	論理演算子
,	カンマ演算子

三項演算子

三項演算子は、右から左に評価します。
三項演算子は、次の1つだけです。

? :　　　　条件演算子

演算子のついたオペランドは式になり、それぞれ単項式、二項式、三項式といいます。

Cの演算では、自動的に型変換を行います。
これは、式のオペランドの型をそろえたり、数値のサイズを拡張したりするものです。
演算子によって行われる変換は、演算子やオペランドの型によって違います。
しかし、整数型や浮動小数点型のオペランドでは、ほぼ同じ規則で変換を行います。
自動的な型変換には、代入と算術演算による2種類があります。

1. 代入の場合、右辺の値は左辺の型に変換されます。
2. 通常の算術変換では、被演算数の中で最も順位の高い型に合わせて変換が行われます。

以下にその規則を示します。

① 被演算数の中に long double 型があれば、もう一方の被演算数も long double 型に変換される。
② ①が成立しない場合、被演算数の中に double 型があれば、もう一方の被演算数も double 型に変換される。
③ ②が成立しない場合、被演算数の中に float 型があれば、もう一方の被演算数も float 型に変換される。
④ ③が成立しない場合、被演算数の中に char 型、unsigned char 型、short 型、unsigned short 型、enum 型、整数のビットフィールドが存在するなら、それらはすべて int 型に変換される。
⑤ そして、被演算数の中に unsigned long int 型があれば、もう一方の被演算数も unsigned long int 型に変換される。
⑥ ⑤が成立しない場合、一方の被演算数が long int 型で、かつ、もう一方の被演算数が unsigned int 型ならば、それらは両方とも unsigned long int 型に変換される。
⑦ ⑥が成立しない場合、被演算数の中に long int 型があれば、もう一方の被演算数も long int 型に変換される。
⑧ ⑦が成立しない場合、被演算数の中に unsigned int 型があれば、もう一方の被演算数も unsigned int 型に変換される。
⑨ ⑧が成立しない場合、被演算数はすべて int 型となる。

2.2.1 算術演算子

算術演算子には、乗算（*）、除算（／）、剰余（%）、加算（+）、減算（−）、マイナス（−）、プラス（+）があります。
マイナス演算子、プラス演算子が単項演算子である他は、すべて二項演算子です。
算術演算子は、オペランドに対して自動的な型変換を行い、演算結果は変換後のオペランドの型になります。
算術演算子によって行う変換には、オーバーフローやアンダーフローは考えられていません。
もし、演算結果が変換後のオペランドの型で表せない範囲の数値のときは、値は確定できません。

乗算演算子（*）

オペランド　*　オペランド

乗算演算子は、2つのオペランドを掛け合わせます。
オペランドの型は、整数型か浮動小数点型です。

```
int x = 10;
double y = 20.0, z;
z = x * y;
```

上の例では、zには200.0が代入されます。
結果の型はdouble型です。

除算演算子（／）

オペランド1 ／ オペランド2

除算演算子は、オペランド1をオペランド2で割ります。
オペランドの型は、整数型か浮動小数点型です。
整数どうしの除算結果が整数でないときは、小数点以下は切り捨てられます。

```
int x = 20, y =6, z;
z = x / y;
```

上の例では除算結果の小数点以下が切り捨てられ、zには3が代入されます。

剰余演算子（%）

オペランド1　%　オペランド2

剰余演算子は、オペランド1をオペランド2で割り、その余りをとります。
オペランドの型は整数型です。

```
int x = 20, y = 6, z;
z = x % y;
```

上の例では、zには2が代入されます。

加算演算子（+）

オペランド　+　オペランド

加算演算子は、2つのオペランドを加えます。
オペランドの型は、整数型か浮動小数点型です。
一方のオペランドがポインタでもかまいません。
ポインタと整数の加算については、「4.2　ポインタ」を参照してください。

```
int x = 20, y = 50, z;
z = x + y;
```

上の例では、zには70が代入されます。

減算演算子（−）

オペランド1 − オペランド2

減算演算子は、オペランド1からオペランド2を引きます。
オペランドの型は、整数型か浮動小数点型です。
減算演算子を使ってポインタ値から整数値を引いたり、ポインタ値からポインタ値を引くこともできます。
ポインタの減算については、「4.2　ポインタ」を参照してください。

```
int x = 100, y = 60, z;
z = x - y;
```

上の例では、zには40が代入されます。

マイナス演算子（－）

－オペランド

マイナス演算子は、オペランドの負の値（2の補数）にします。
オペランドの型は、整数型か浮動小数点型です。

プラス演算子（＋）

＋オペランド

プラス演算子は、マイナス演算子と対になるもので特に演算作用はありません。

2.2.2 代入演算子

Cでは、代入を式として扱います。
したがって、代入は演算の1つと考えられ、いくつかの代入演算子が用意されています。
代入演算子は、次の通りです。

＋＋	インクリメント演算子
－－	デクリメント演算子
＝	単純代入演算子
＊＝	乗算代入演算子
／＝	除算代入演算子
％＝	剰余代入演算子
＋＝	加算代入演算子
－＝	減算代入演算子
＜＜＝	左シフト代入演算子
＞＞＝	右シフト代入演算子
＆＝	ビットごとのAND代入演算子
｜＝	ビットごとのOR代入演算子
＾＝	ビットごとの排他的OR代入演算子

代入によって、右辺値の型は左辺値の型に変換されます。
代入は、左辺オペランドのアドレスに右辺オペランドの値を格納することです。

したがって、代入演算の左辺オペランドや単項代入式のオペランドは、アドレスを参照する式を指定します。
このような式を左辺値式といいます。
変数名は、その変数の値のアドレスを指しているので、左辺値式になります。
左辺値式にある式は、次の通りです。

- ●文字型、整数型、浮動小数点型、ポインタ型、列挙型、構造体型、共用体型の識別子
- ●添字式（結果が配列を指すポインタにならないもの）
- ●メンバ参照式
- ●間接演算子の式（配列を参照しないもの）
- ●ポインタ型への型キャスト式
- ●カッコで囲まれた左辺値

なお、代入演算子は単一のトークンなので、演算記号の間に空白を入れてはいけません。

インクリメント／デクリメント演算子

インクリメント／デクリメント演算子は、オペランドの値を増加、減少します。
オペランドの型は、整数型か浮動小数点型、またはポインタ型です。
整数型か浮動小数点型のオペランドでは、増減の幅は整数値１です。
ポインタ型のオペランドでは、ポインタを１つ進めるか、１つ戻します。
ポインタについては、「4.2　ポインタ」を参照してください。
この演算子は、オペランドの前にも後にも置けます。
前に置いたときと後に置いたときの違いを、インクリメント演算子を例に説明します。

```
int x, y, z ;
x = 10 ;
y = ++x ;
x = 10 ;
z = x++ ;
```

ｘを１インクリメント（増加）した値がｙに代入されるので、ｙの値は11になります。
これに対し、インクリメント演算子をオペランドの後に置くと、代入が先に実行されます。

そのため、z の値は 10 になります。

単純代入演算子

単純代入演算子（=）は、右のオペランドを左のオペランドに代入します。
代入時の型の変換については、「2. 4　型変換」を参照してください。

```
double x;
int y;
x = y;
```

上の例では、y の値を double 型に変換して x に代入します。

複合代入演算子

複合代入演算子は、単純代入演算子（=）と二項演算子を組み合わせたものです。
オペランドの型は、それぞれの二項演算子で使える型です。

```
オペランド1　+=　オペランド2
```

上の式は、次のように解釈されます。

```
オペランド1　=（オペランド1）+（オペランド2）
```

ただし、複合代入演算子を使うと、オペランド1は1回しか評価されません。

```
x = 10;
x *= 3;
```

上の例では、x には 30 が代入されます。

```
x = 10;
1.  x  += (++x);
2.  x = (++x) + (++x);
```

上の1の式の結果は、2の式とは違ったものになります。

2.2.3 関係演算子

関係演算子は、2つのオペランドを評価し、演算子で指定した関係を調べます。
オペランドの型は、整数型か浮動小数点型、またはポインタです。
結果は整数型で、真のときは1、偽のときは0になります。
関係演算子は、次の関係を評価します。

オペランド1<オペランド2	オペランド1は、オペランド2より小さい
オペランド1>オペランド2	オペランド1は、オペランド2より大きい
オペランド1<＝オペランド2	オペランド1は、オペランド2より小さいか等しい
オペランド1>＝オペランド2	オペランド1は、オペランド2より大きいか等しい
オペランド1＝＝オペランド2	オペランド1とオペランド2は等しい
オペランド1!＝オペランド2	オペランド1とオペランド2は等しくない

演算子＝＝と!＝のオペランドには、列挙型を使うこともできます。
ポインタについては、「4.2　ポインタ」を参照してください。

```
int x =10, y = 10;
```

```
例1
  x<y  /* false  */

例2
  x>y   / false  */

例3
  x<=y  /* true  */

例4
  x>=y  /* true  */
```

例5

```
x==y  /* true  */
```

例6

```
x !=y  /* false  */
```

上の例では、x と y とは等しいので、例3、4、5は1に、例1、2、6は0になります。

2.2.4 論理演算子

論理演算子には、二項演算子の AND (&&)と OR (||)、単項演算子の NOT (!)があります。
オペランドの型は、整数型か浮動小数点型、またはポインタです。
論理演算子は、各オペランドが0かどうかについて評価します。
結果は整数型で、真のときは1、偽のときは0になります。

論理的 AND 演算子 (&&)

オペランド1　&&　オペランド2

論理的AND演算子は、オペランドが両方とも0でないとき真となり、結果は1になります。
どちらか一方のオペランドが0のときは偽となり、結果は0です。
もし、オペランド1が0となったとき、評価はそこで打ち切られ、オペランド2は評価しません。

論理的 OR 演算子 (||)

オペランド1　||　オペランド2

論理的OR演算子は、どちらか一方のオペランドが0でないとき真になり、結果は1になります。
両方とも0のときは偽となり、結果は0です。
もし、オペランド1が0でないと評価は打ち切られ、オペランド2は評価されません。

論理的 NOT 演算子（！）

！ オペランド

論理的 NOT 演算子は、オペランドの条件を反転します。
オペランドが 0 のときは、そのオペランドの評価は偽なので、反転して真となり、結果は 1 です。
オペランドが 0 でないときは、オペランドの評価は真なので、演算結果は偽となり、値は 0 です。

```
int x = 50, y = 30, z = 20;
```

例 1

```
x < y && y < z
```

例 2

```
x < z || z < y
```

例 3

```
!( x == y )
```

例 1 では、x<y が偽なので結果は 0 です。
y<z は評価されません。
例 2 では、x<z が偽なので z<y を評価します。
z<y は真ですので、結果は 1 です。
例 3 の x==y は偽です。
評価が反転されて結果は 1 です。

2.2.5 条件演算子

条件演算子は三項演算子で、次のもの 1 つだけです。

オペランド1?オペランド2：　オペランド3

オペランド1の型は、整数型か浮動小数点型、またはポインタです。
結果の型は、オペランド2か3の型です。
オペランド2、オペランド3は同じものであれば、構造をもった型を置くこともできます。
まず、オペランド1を評価します。
結果が真（0でない)であれば、オペランド2を評価し、その結果を式の値とします。
オペランド1が偽（0）のときは、オペランド3を評価し、その結果が式の値となります。
オペランド2とオペランド3の両方を評価することはありません。

```
int x, y, z;
z = (x>y)? x : y;
```

上の例では、xの値とyの値のどちらか大きいほうがzに代入されます。

2.2.6 間接演算子とアドレス演算子

間接演算子（*）

＊オペランド

間接演算子は、ポインタが指すアドレスの内容を参照します。
オペランドの型はポインタです。
結果の型は、オペランドが指す値の型になります。
ポインタの値がヌルのとき（ヌルポインタ）は、ポインタはどのアドレスも参照していません。

アドレス演算子（&）

&オペランド

アドレス演算子は、オペランドのアドレスを返します。
オペランドには、代入演算の左辺値式として使えるものを指定します。
結果の型は、オペランドのポインタです。
アドレス演算子は、構造体のビットフィールドメンバと、register宣言され

ている識別子は使えません。

```
int *ptr;
int arry[10];
```

例1

```
ptr = &arry[2];
```

例2

```
x = *ptr;
```

例1では、アドレス演算子は、配列 arry の2番目の要素のアドレスを返し、結果をポインタ変数 ptr に代入します。
例2では、間接演算子は、変数 ptr のアドレスにある整数型の値を参照して、結果を x に代入します。

2.2.7 シフト演算子

シフト演算子は、オペランド2が指定するビット数だけオペランド1を左（<<）か右（>>）にシフトします。

オペランド1 $\left\{ \begin{matrix} << \\ >> \end{matrix} \right\}$ オペランド2

オペランドの型は、両方とも整数型です。
結果の型は、自動的な型変換後のオペランドの型です。
左シフトされて空いた右ビットには、0が入ります。
右シフトで、オペランド1の型が unsigned 型ならば、空いた左ビットには0が入ります。
オペランド1がそれ以外の型のときは、符号ビットのコピーが入ります。
もし、オペランド2の値が負のときには、結果は不定になります。
シフト演算子によって行う変換には、オーバーフローやアンダーフローは考えられていません。

もし、演算結果が変換後のオペランド1の型で表せない範囲の数値のときは、値は確定できません。

```
unsigned int a, b, c ;
a = 0x00FF ;
b = 0x1100 ;
c = (a<<8) + (b>>8) ;
```

上の例では、a は左に8ビット分シフトされ、b は右に8ビット分シフトされます。
c には、0xFF11 が導入されます。

2.2.8 sizeof 演算子

sizeof 演算子は, 識別子や型に割りあてられている記憶領域のバイト数を返します。

```
sizeof (名前)
```

カッコの中の名前には、識別子か void 型以外の型名を指定します。
返される式の値は、指定した識別子や型に割りあてられているバイト数です。
sizeof 演算子の名前に構造をもつ型の型名や識別子を指定すると、結果はその型のもつ全体のバイト数になります。
ただし、構造体や共用体のメンバをメモリの境界に合せるために使用するパディング部分も、バイト数に含むことがあります。
そのようなときは、結果と各メンバの合計バイト数とが一致しません。

```
int x, arry [5] ;
x = sizeof (arry) ;
```

上の例では、XC コンパイラの int 型が4バイトなので、x には20が代入されます。

2.2.9 ビット演算子

ビット演算子には、二項演算子のビットごとのAND (&)、OR (|)、排他的OR (^) と、単項演算子のビットごとの補数 (¯) があります。
オペランドの型は整数です。
結果の型は、自動的な型変換後のオペランドの型です。

ビットごとのAND演算子 (&)

オペランド1 & オペランド2

ビットごとのAND演算子は、オペランド1の各ビットとそれに対応するオペランド2のビットを比較して、ビットのANDをとります。
比較されたビットが両方とも1のときだけ、対応する結果のビットを1にセットし、それ以外のときは、対応する結果のビットを0にセットします。

ビットごとのOR演算子 (|)

オペランド1 | オペランド2

ビットごとのOR演算子は、オペランド1の各ビットとそれに対応するオペランド2のビットを比較して、ビットORをとります。比較されたビットが両方とも0のときだけ、対応する結果のビットを0にセットします。
それ以外のときは、対応する結果のビットを1にセットします。

ビットごとの排他的OR演算子 (^)

オペランド1 ^ オペランド2

ビットごとに排他的OR (XORともいいます) 演算子は、オペランド1の各ビットとそれに対応するオペランド2のビットを比較して、ビットのXORをとります。
比較されたビットが同じでなければ、それに対応する結果のビットを1にセットします。
それ以外のときは、対応する結果のビットを0にします。

ビットごとの補数演算子 (¯)

¯オペランド

2.2 演算子

ビットごとの補数演算子は、そのオペランドのビットを反転します。
ビットが1のとき0に、0のとき1にセットします。

各ビットごとの演算の結果を、表にまとめておきます。

オペランド		op1&op2	op1 | op2	op1 ^ op2	~op1
op1	op2				
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

注）~op1 はオペランド op1 に対するビットごとの補数演算です。

```
short x = 0x12AB;
short y = 0x1200;
short z;
```

例1
```
z = x & y;
```

例2
```
z = x | y;
```

例3
```
z = x^y;
```

例4
```
z = ~x;
```

例1では、ビットごとの AND をとり、z には 0x1200 が代入されます。
例2では、ビットごとの OR をとり、z には 0x12AB が代入されます。
例3では、ビットごとの XOR をとり、z には 0x00AB が代入されます。
例4では、ビットを反転するので、z には 0xED54 が代入されます。

2.2.10 カンマ演算子

カンマ演算子は、オペランドを左から右に順に評価します。
演算の結果は、一番右のオペランドの値と型になります。
オペランドの型に制限はありません。
この演算子は、1つの式しか書けないところで、複数の式を評価するときに使用します。
ただし、for 文の式を区切ること以外にはほとんど使いません。

```
for (op1 = op2 = 0 ; op1 <100 ; op1++, op2--)
```

例の for 文のループ式（op1++と op2--）は、別々に評価します。
左の op1++を先に評価し、その後 op2--を評価します。

2.3 演算子の優先順位

Ｃの演算子には、評価の優先順位と結合規則があります。
式の中では、優先順位の高い演算子から先に評価します。
同じ優先順位の演算子が並んでいるときには、結合規則により評価の順位が決定されます。

2.3.1 評価優先順位と結合規則

下の表は、Ｃの演算子の優先順位と結合規則を要約したものです。
表は優先順位の高い順に、上から下へ並んでいます。

演算子		結合規則
（ ）　［ ］　．　－＞	式	左から右
＋　－　~　！　＊　＆ ＋＋　－－　sizeof　キャスト	単項	右から左
＊　／　％	剰余算	左から右
＋　－	加減算	左から右
＜＜　＞＞	シフト	左から右
＜　＞　＜＝　＞＝	関係（非等値性）	左から右
＝＝　!＝	関係（等値性）	左から右
＆	ビット AND	左から右
＾	ビット EOR	左から右
|	ビット OR	左から右
＆＆	論理 AND	左から右
‖	論理 OR	左から右
？　：	条件	右から左
＝　＊＝　／＝　％＝ ＋＝　－＝ ＜＜＝　＞＞＝ ＆＝　|＝　＾＝	単純代入／複合代入	右から左
，	逐次評価	左から右

2.3 演算子の優先順位

同じ優先順位をもつ演算子（上記の表で同じ行のもの）が、式の中に同じレベルで記述されているときは、結合規則に従って右から左へ、または左から右へ演算子を評価します。

オペランドを入れ替えても結果の変わらない演算子（乗算演算子、加算演算子、二項ビット演算子）が同じレベルで続いているときの評価の順序は、決められていません。

カンマ演算子、論理的 AND 演算子、論理的 OR 演算子が同じレベルで続いたときは、必ず結合規則の方法どおりに左から右に評価します。

しかし、論理演算子は式の結果が確定したところで評価を打ち切るので、式の中のすべてのオペランドを評価しないことがあります。

```
x < y && y < z ||a ++
⇨ ((x < y) && (y < z)) ||a++
```

上の例では、論理 AND 演算子の優先順位が論理的 OR 演算子より高いので、右の式のようにカッコの中を１つのオペランドと考えます。

論理演算子は、必ずオペランドを左から右に評価するので、論理的 OR 演算はカッコ中を先に評価します。

ここでカッコの中が真と評価されると、論理 OR 演算の評価は打ち切られ、a はインクリメントされません。

これを避けるには、式を a＋＋から始めるか、a のインクリメントを別の式で行います。

例 1

```
x == a ? x += 1 : x += 2
➜(x == a ? x += 1 : x) += 2
```

例 2

```
(X == a) ? (x += 1) : (x += 2)
```

上の例 1 は、優先順位から右の式のように評価されます。

このため、＋=2 には右辺オペランドがなく、エラーになります。

例 2 のようにカッコを使えば、エラーは防ぐことができます。

2.3.2 演算の副作用

副作用とは、式の評価の結果や状態によって値が変化することをいい、変数の値の変化にかかわりなく発生します。
代入演算や代入演算を引数にもつ関数の呼び出しでは、副作用を発生する場合があります。
副作用の評価の順序は、定義されていません。

例 1

```
int x = 3, y = 4;
y= x / ++x;
```

例 2

```
arr [i] = i ++;
```

上の例は、どちらもどのように評価されるか不明です。
例 1 では、除算の分子である x の値は 3 か 4 です。
例 2 では、配列の添字の値がインクリメントする前か後かで、結果は異なったものになります。

2.4 型変換

型変換には、次のものがあります。

- 型の違う変数へ代入するときの変換
- 他の型へのキャスト
- 演算子による自動的変換
- 引数をもつ関数の呼び出しによる変換

2.4.1 代入による変換

代入演算では、代入する値の型を代入先の変数の型に変換します。
変換によって内容が保証されない（値が確定できない）ことがあっても変換します。

符号つき整数型の変換

符号つき整数は、短いサイズへの変換では上位ビットを切り捨てます。
長いものへ変換するときは、符号拡張をします。
符号つき整数から浮動小数点数へ変換するときは、内容は保証されますが、もし、整数がint型、またはlong型だと精度が失われることがあります。
符号つき整数を符号なし整数に変換すると、符号なし整数のサイズに変えられ、符号ビットは無意味になります。

符号なし整数型の変換

符号なし整数は、短いサイズの変換では上位ビットを切り捨てます。
長いものへ変換するときは、ゼロ拡張します。
符号なしの整数の値を浮動小数点数に変換するときは、まずlong型に変換し、さらにその値を浮動小数点数に変換します。
符号なし整数を符号つき整数に変換したときに、サイズが同じであればビットパターンは変化しませんが、符号ビットをセットすると値は変化します。

浮動小数点型の変換

float型をlong double型やdouble型に変換しても、値は変わりません。
long double型やdouble型からfloat型に変換するときには、値がfloat型

で表せる範囲を超えると、精度が失うか内容が保証されなくなります。
float型を整数型に変換するときには、いったんlong型に変換し、さらに必要であれば他の整数型に変換します。
long double型やdouble型を整数型に変換する場合、整数型がchar型やshort型のときは、float型・long型と変換してからchar型やshort型に変換します。
それ以外の整数型のときは、直接long型に変換してから、必要なら他の整数型に変換します。
浮動小数点数の小数部分は、long型への変換で切り捨てられます。
long型で表せる範囲を超えると、内容は保証されません。

その他の型の変換

enum型の値は、整数として定義されているので、変換はint型として行います。
構造体型や共用体型の型変換はできません。
ポインタ型を型変換の対象にしたとき、ポインタはint型の符号なし整数として扱います。
ただし、ポインタは浮動小数点型には変換できません。
void型は値をもたないので、代入によってvoid型を他の型に変換したり、他の型をvoid型に変換することはできません。
基本型の代入変換規則表を次頁に示します。

2.4 型変換

基本型の代入変換規則表

To \ From	char	short	int	long	unsigned char	unsigned short	unsigned int	unsigned long	float	double	long double
char		下位8bit	下位8bit	下位8bit	(符号ビット有)	下位8bit	下位8bit	下位8bit	↓long↓	↓float↓	↓float↓
short	符号拡張		下位16bit	下位16bit	ゼロ拡張	(符号ビット有)	下位16bit	下位16bit	↓long↓	↓float↓	↓float↓
int	符号拡張	符号拡張		○	ゼロ拡張	ゼロ拡張	(符号ビット有)	(符号ビット有)	↓long↓	↓long↓	↓long↓
long	符号拡張	符号拡張	○		ゼロ拡張	ゼロ拡張	(符号ビット有)	(符号ビット有)	↓long↓	↓long↓	↓long↓
unsigned char	(符号ビット無し)	下位8bit	下位8bit	下位8bit		下位8bit	下位8bit	下位8bit	↓long↓	↓long↓	↓long↓
unsigned short	↓short↓	(符号ビット無し)	下位16bit	下位16bit	ゼロ拡張		下位16bit	下位16bit	↓long↓	↓long↓	↓long↓
unsigned int	↓long↓	↓long↓	(符号ビット無し)	(符号ビット無し)	ゼロ拡張	ゼロ拡張		○	↓long↓	↓long↓	↓long↓
unsigned long	↓long↓	↓long↓	(符号ビット無し)	(符号ビット無し)	ゼロ拡張	ゼロ拡張	○		↓long↓	↓long↓	↓long↓
float	↓long↓	↓long↓	注1)	注1)	↓long↓	↓long↓	↓long↓	↓long↓		注2)	注2)
double	↓long↓	↓long↓	注1)	注1)	↓long↓	↓long↓	↓long↓	↓long↓	(内部表現の変化)		○
long double	↓long↓	↓long↓	注1)	注1)	↓long↓	↓long↓	↓long↓	↓long↓	(内部表現の変化)	○	

注1) int型又はlong型で表される整数の場合、変換後の精度が失われることがあります。

注2) 精度が失われるか、内容が保障されなくなります。

左頁表の枠内の○は、値の精度はまったく失われず変換され、内容が保証されることを表します。
また、表の枠の中に long 型や float 型といった基本型が記入されている場合は、いったんその型に変換されることを表します。
たとえば、double 型から char 型への変換では、

double 型　→　float 型　→　long 型

という順序で変換され、さらに long 型の値の下位 8bit 分が char 型に代入される、ということを意味します。
なお、int 型と long 型、unsigned int 型と unsigned long 型、long double 型と double 型は、それぞれ同じ大きさであり、その扱いは同じです。

2.4.2 型キャスト変換

型のキャストを使って、強制的な型変換を行えます。
型キャストの書式は、次の通りです。

(型名) オペランド

代入による変換の規則は、型キャストにも適用されます。
void 型を、型キャストの型名として使用できますが、結果はどのような項目にも代入できません。

第3章

制御構造と関数

文とブロック

関　数

本章では、C 言語の文と関数について説明します。

文とは、C プログラム中での処理や制御構造を決めるもので、必ずセミコロン（;）で終わります。
C プログラム中では、文は基本的に記述順序に従って実行されますが、条件つき、または無条件に他の位置に制御を移す場合もあります。

関数とは、処理を実行するためのサブルーチンで、文で構成されています。
C プログラムでは、最低 1 つの関数 main が必要で、プログラムの実行は必ず main から始まります。

3.1 文とブロック

文には、次の種類があります。

複文	式文	空文	break 文
continue 文	do-while 文	for 文	goto 文
if 文	return 文	switch 文	while 文

3.1.1 空文、式文、複文

●空　文

空文とは、文本体や式をともなわないセミコロン（；）のみの文です。
空文は、プログラムの処理には何も影響を与えません。
文法を満たすために、文が必要な場合に記述します。

書　式

```
;
```

プログラム例

```
for (i=0; i<maxline; str[i++]=' ')
        ;
```

上の例は、str を空白で満たすものですが、その処理は for 文の 3 番目の式 str［i＋＋］＝' 'で実行されています。
しかし、for 文は実行可能な文を文法上必要としていますので、空文を使用しています。
空文の前にもラベルをつけることができます。

●式　文

式文とは、式の末尾にセミコロン（；）をつけて文にしたものです。

書　式

```
式;
```

例 1
```
x=(y * 3);
```

例 2
```
i++;
```

例 3
```
func (a);
```

例 4
```
y=func (a);
```

例 1 は、左辺値 y * 3 を x に代入します。
例 2 は、i の値を 1 つインクリメントします。
例 3 は、関数の呼び出し式で、a は引数です。
関数の戻り値を利用する場合は、例 4 のように記述します。

●複　文

複文とは、式文や空文などの複数の文で構成される文で、他の文の本体として記述されます。

書　式

```
{
  [宣　言]
    ・
    ・
    ・
  文　1
  [文　2]
    ・
    ・
    ・
}
```

［と］の中は、省略することが可能です。

プログラム例

```
if (i <= 100){
        str[i] = a;       /*  100以下の場合   */
        a += 2;           /*  この3行が実行   */
        i++;              /*  されます.       */
}
```

上の例では i が 100 以下のとき、中カッコ({ }) で囲まれたブロックを上から順に実行します。
複文にはラベルをつけることができます。

3.1.2 break 文

break 文は、break のあるネストの do-while、for、switch、while 文の実行を終了させ、制御を直後のブロックに移します。
ネスト全体から抜け出したい場合は、goto 文または return 文を使います。
break 文は、do-while、for、switch、while 文の外側では記述できません。

書　式

break ;

プログラム例

```
for ( i = 0 ; i < MAXLEN ; i++ ) {
        for ( j = 0 ; j < MAXWIDTH ; j++ ) {
                if ( str[i][j] == NULL ) {     /*  str[i][j]はヌルか?   */
                        len[i] = j;      /*  ヌルであれば長を格納  */
                        break;           /*  jのループを抜ける    */
                }
        }       /*  jのループの終り   */
}       /*  iのループの終り   */
```

上の例では、文字列 str［i］［j］の末尾のヌル文字(NULL＝'¥0')を発見すると、文字列の長さを len[i］に格納した後に break 文を実行します。
break 文が実行されると、内側の for のループから外側の for のループに制御が移り、i がインクリメントされます。

3.1.3 continue 文

continue 文は、do-while、for、while 文のループ内で、continue 文の次のブロックを実行せずに、次の繰り返しに制御を移します。

for 文の場合： ループ式（「3. 1. 5　for 文」参照）に制御を移します。

do-while 文、while 文の場合： while の式（「3. 1. 4　do-while 文」、「3. 1. 10　while 文」参照）に制御を移します。

書　式

continue ;

プログラム例

```
while ( i++ < MAX ) {
        a = func1(i);
        if (a == 2)
                continue;
        b = func2(i);
}
```

関数 func1 の戻り値が 2 であれば、while のループを繰り返します(func2 は実行しません)。
関数 func1 の戻り値が 2 以外の場合のみ、func2 を実行して while のループを繰り返します。

3.1.4 do-while 文

do-while 文は、while 後の式が真である間、文を実行します。
文を実行した後に式が評価されますので、文は最低 1 回は実行されます。
式が偽になると do-while 文を終了し、次の文に制御を移します。
式が真であっても、do-while 文中に break、goto、return 文を記述することにより終了させることもできます。

書　式

```
do
        文
while（式）；
```

プログラム例

```
do {
        b = func(a);
        a++;
} while ( a < 127 );
```

上の例では、まず関数 func を実行し、戻り値を b に代入し、a の値を 1 つインクリメントします。
その後に a の値を評価し、127 未満であればループを繰り返し実行します。

3.1.5 for 文

for 文は、条件式が真である間、文を実行します。
条件式が偽 (0) になると、次の文に制御を移します。
条件式が真であっても、for 文中に break、goto、return 文を記述することにより終了させることもできます。
for 文の処理の流れは、次の通りです。

①初期化式を実行します。
②条件式を評価します。
　真であれば③に進みます。
　偽であればループを終了し、次の文に制御を移します。
③文を実行します。
④ループ式を実行し、②に戻ります。

書　式

for(［初期化式］；［条件式］；［ループ式］)
　　　文

［と］で囲まれた初期化式、条件式、ループ式は省略が可能です。
ただし、セミコロン（；)は省略できません。
また、条件式を省略した場合は、常に条件式は真と解釈されます。

プログラム例

```
for ( i = 0; i < MAX; i++ ) {
        if ((str[i] >= 'a') && (str[i] <= 'z'))
                str[i] = str[i] - 32;     /*  大文字 = 小文字 - 32  */
}
```

上の例では、iを0から1ずつインクリメントし、MAX 未満の範囲で文字列 str 内の文字を検索します。
文字が英小文字であれば、英大文字に置き換えます。

3.1.6 goto 文

goto 文は、制御を無条件にラベルで指定する文に移します。
goto 文のラベルの末尾は、必ず（：)でなくてはなりません。
また、指定したラベルは、同一関数内に存在しなくてはなりません。
ラベルは、goto 文に対応する以外には、まったくプログラムの実行には影響しません。

書　式

```
goto ラベル；
    ・
    ・
    ・
ラベル：文
```

プログラム例

```
if (errflg > 0)
        goto exit;
                .
                .
                .
exit: fprintf(stderr,"入力された値が範囲外です．¥n");
        return(errflg);
```

上の例では、エラーが発生した場合 exit というラベルのついた文に制御を移し、エラーメッセージを表示します。

3.1.7 if 文

if 文は条件式の評価結果により実行する文を選択します。
条件式が真のとき文 1 を実行します。
else があった場合、文 2 は実行しません。
条件式が偽のときは、if 文の次の文を実行します。
else があれば文 2 を実行した後に if 文の次の文を実行します。

3.1 文とブロック

書　式

if（条件式）
　　　文 1
［else
　　　文 2］

［と］の中は省略が可能です。

プログラム例

```
if (i > 0)
        b = a * i;
else
        b = a * (-i);
```

上の例では、i が正のときは b=a ＊ i を実行し、i が 0 または負のときは b=a ＊(−i) を実行します。
if 文や else 文に別の if 文を記述してネストすることもできます。
この場合、中カッコ（｛　｝）で囲んでブロックを明確に記述してください。

例 1

```
if (i > 0)
        if (j > i)
                x = j;
        else
                x = i;
```

例 2

```
if (i > 0) {
        if (j > i)
                x = j;
}
else
        x = i;
```

例 1 では、else は内側の if 文に対応しますが、例 2 では、外側の if 文を中カッコで囲ってあるので、else は外側の if 文に対応しています。

3.1.8 return 文

return 文は、現在実行中の関数を終了し、呼び出しもとの関数に制御を移します。
呼び出しもとでは、関数の呼び出しを行った直後の文から実行を再開します。
return 文の式が関数の戻り値として、呼び出しもと関数にわたされます。
式を省略すると、戻り値は不定になります。

書　式

return［式］；

［と］の中は、省略が可能です。

プログラム例

```
main()
{
int     func1(int);
void    func2(int,int);
        .
        .
        .
        y = func1(x);
        func2 (x, y);
        .
        .
        .
}
int func1(int a)
{
        return((a) * (a));
}
void func2(int a, int b)
{
        .
        .
        .
        return;
}
```

main 関数から呼び出された func1 関数は、return 文の式 (a) * (a) により a の二乗を返します。
この戻り値は y に代入されます。
func2 の戻り値は、main 関数内で利用されていませんので、func2 関数は void 型で定義され、return 文の式も省略されています。

3.1.9 switch 文

switch 文は、式の値と一致するラベルをもつ case 以降に制御を移します。

case 以降の処理をすべて終了するか break 文を実行することよにより、switch 文から抜け出します。

switch 文の式と case 分の定数式は、ともに整数型でなくてはなりません。

書　式

switch（式は整数型のみ、enum も） {
　　　　［宣言］
　　　　　⋮
　　　　［case　定数式：］
　　　　　⋮
　　　　　　　　［文］
　　　　　　　　　⋮
　　　　［default：
　　　　　　　　［文］］
}

［と］の中は、省略が可能です。

プログラム例

```
switch (c) {
        case NUM:
                number++;
        case LET:
                letter++;
        default:
                counter++;
}
```

switch の式 c の値が NUM に等しいときは、number＋＋、letter＋＋および counter＋＋を実行します。

c の値が LET に等しいときは、letter＋＋および counter＋＋を実行します。

c の値が NUM とも LET とも等しくないときは、default の counter＋＋のみを実行します。

プログラム例

```
switch (flg) {
        case ON:
                x++;
                break;
        case OFF:
                y++;
                break;
        case ERR:
                zz++;
                break;
}
```

上の例では、flg が ON のとき x をインクリメントしますが、break 文により以降の文は無視して switch 文を終了します。

プログラム例

```
swith (c) {
        case 1:
        case 2:
        case 3:
        case 4: counter++;
}
```

上の例では、1 つの文に case　1 から case　4 の 4 つのラベルがついています。

この場合、c が 1 から 4 のいずれかと一致すれば、counter＋＋が実行されます。

3.1.10 while 文

while 文は、式の値が真（0でない）であれば文が実行され、実行後、式を再評価します。
式が最初から偽 (0) のときは、while 文の本体は一度も実行されず、制御は while 文の次の文へ移ります。
while 文は、条件が真であっても、文の本体内の break、goto、return 文の実行によってループを終了させることもできます。

書　式

while（式）
　　文

プログラム例

```
while (i >= 0) {
        str1[i] = str2[i];
        i--;
}
```

例では、i の値が 0 以上の間、while 文の本体が繰り返され、str2 が str1 にコピーされます。

3.2 関　数

関数とは、特定の処理を実行するためのサブルーチンです。
C 言語ではプログラムは関数の集合として記述されます。

3.2.1 関数の定義

関数は、関数名、仮引数、引数の宣言、関数の動作を記述した文を指定して定義します。
関数の戻り値と記憶クラスも指定できます。
関数を宣言するときは、他の関数と明確に区別できるように関数の名前、戻り値の型、記憶クラスを設定しなければなりません。
コンパイラは、関数の引数の型をチェックしません。
チェックするには、プロトタイプ宣言を行います。
関数の戻り値の型が int 型のときは、戻り値の型の宣言は省略できます。

書　式

［記憶クラス指定子］［型指定子］宣言子（［引数宣言リスト］）
関数本体

［と］の中は、省略が可能です。

記憶クラス指定子

記憶クラス指定子は、関数の記憶クラスを指定します。
指定できる記憶クラスは、extern または static です。
指定を省略すると、関数の記憶クラスは extern になります。
extern 記憶クラスの関数は、ソースプログラム中のどこからでも呼び出せ、static 記憶クラスの関数は、同じソースファイル内でのみ呼び出せます。
すでに、extern 記憶クラスとして宣言されている関数を static として定義すると、その関数の記憶クラスは static になります。

型指定子と宣言子

型指定子は、関数の戻り値の型を指定します。
基本型の他、構造体型や共用体型を指定できます。

指定を省略すると、int 型になります。
宣言子は、関数の名前を定義します。
関数名にアスタリスク(*)をつけると、ポインタを返します。
関数の定義で指定する戻り値の型は、宣言時のその関数の戻り値の型と同じでなければなりません。
int 型以外の戻り値の型を持つ関数は、定義前や宣言前に呼び出せません。
また、型指定子と宣言子をまとめて、新たな型として typedef 記憶クラス指定子で定義することはできません。
関数が式をもつ return 文を実行すると、式を評価し、必要に応じて戻り値の型に変換して値を返します。
return 文を実行しないか、実行して return 文に式が含まれていない場合は、関数の戻り値を呼び出し側の関数で使うことはできません。
また、void 型で定義した関数は、何の値も返しません。

プログラム例

```
static power2(x)
int x;
{
        return((x) * (x));
}
```

上記の関数は static です。
型指定子が省略されていますので、戻り値は int 型です。

プログラム例

```
typedef struct {
        char name[30];
        int age;
} PERSON;

PERSON sel_young(per1, per2)
PERSON per1,  per2;
{
        return ((per1.age < per2.age) ? per1 : per2);
}
```

typedef で PERSON 型を定義し、関数 sel_young は PERSON 型の戻り値を返します。

プログラム例

```
char * smalls(str1, str2)
char *str1 , *str2;
{
        int i;

        i = 0;
        while ((str1[i] != NULL) && (str2[i] != NULL))
                i++;
        return((str1[i] == NULL) ? str1 : str2);
}
```

関数 smalls は、文字例へのポインタを返します。

仮引数リストの宣言

仮引数とは、関数が呼び出されたときに、関数にわたされる値を受けとる変数です。

仮引数リストの宣言は、仮引数の型と名前及び関数に値をとり込む順番を定義し、カッコ内にカンマ(,)で区切って記述します。

仮引数の記憶クラス指定子は、register のみ指定でき、省略すると記憶クラスは auto になります。

関数本体

関数本体は、ローカル変数の宣言と文で構成されます。

関数は、文の最後まで実行するか return 文を実行すると終了し、呼び出しもとに制御を戻します。

return 文を実行しなかったり、return 文が関数本体内にない場合は、関数の戻り値は不定になります。

3.2.2 関数の宣言

関数を定義する前に宣言することを、フォワード宣言といいます。

フォワード宣言は、関数の属性を設定するものなので、関数の本体や仮引数は定義しません。

記憶クラスを static でとる関数や、戻り値の型が int でないものは、必ずフォワード宣言します。

記憶クラスが extern で戻り値の型が int 型の関数は、フォワード宣言を省略できます。

3.2 関　数

フォワード宣言を省略したときや、引数型リスト省略したときは、コンパイラは引数の型や個数のチェックをしません。
型を変換する必要があると、自動的に型変換が行われます。
引数型リストを指定したフォワード宣言を特に、プロトタイプ宣言といいます。
デフォルトでは、プロトタイプ宣言を必要とし、引数の型や数の不一致に対してコンパイラは警告メッセージを出します。
くわしくは、「Cライブラリマニュアル VOL.1」を参照してください。
プロトタイプ宣言では、引数型に void 型を指定できます。
これは、明らかに引数をもたないことを宣言するものです。
また、void ＊と指定すると、すべてのポインタ型を引数としてとることができます。
引数型はカンマで区切って並べますが、リストの終わりがカンマだけのときには、指定した数より多い引数に対して何のチェックもされません。
関数の戻り値の型として、配列や関数は指定できません。
しかし、戻り値の型をポインタ型にすることで、配列や関数を指すポインタを返すことができます。

書　式

［型指定子］宣言子（［引数型リスト］）

［と］の中は、省略できることを表します。

例 1

```
extern int power (int, int);
```

例 2

```
char *find_arg (char *, int);
```

例 3

```
double div (double, double);
```

例4

```
void clears (void);
```

例1は、もし、引数のチェックを行わなければ省略してもかまいません。
例2の関数は、char 型を指すポインタを返します。
例3では、戻り値の型がdouble型なので必ずフォワード宣言します。
例4は、戻り値も引数もない関数です。

引数リストの最後に省略記号である",..."をつけて関数を宣言すると、明示してある引数より多い引数を指定して、その関数を呼び出すことができます。
このとき、明示されていない追加分の引数を参照するには、ヘッダファイル"stdarg. h"で定義されているva_argマクロを使用する必要があります。
詳しくは、「Cライブラリマニュアル」va_start, va_arg, va_endを参照してください。
また、このように可変個の引数をもつ関数は、その定義において1つ以上の名前付き引数をもつ必要があります。

3.2.3 関数の呼び出し

関数を呼び出すときに与える引数を、実引数といいます。
実引数として与えることのできる型は、戻り値として使える型と同じものです。
したがって、配列や関数を引数としてわたすときには、それらを指すポインタをわたすようにします。
C言語では、引数はすべて値で受けわたします。
これを、値による呼び出し(call by value)といいます。
呼ばれた関数は、実引数のコピーを仮引数に代入して使うので、直接実引数の値を変えられません。
ただし、これは欠点ではなく、関数の独立性を高める有益な手段です。
呼び出し側の変数を変更するときには、ポインタを使って間接的に行います。

3.2 関　数

例 1

```
power(a, n)
int a, n;
{
        int x;
        for ( x = 1; n > 0; --n )
                x *= a;
        return(x);
}
```

例 2

```
main()
{
        int a = 5, b = 10;
        swap(&a, &b);
}

swap(x, y)
int * x, * y;
{
        int p;
         p = *x;
        *x = *y;
        *y = p;
}
```

例 1 の仮引数 n は、関数内でデクリメントされ、最終的にはその値は 0 になります。

しかし、呼び出した側の実引数にはまったく影響しません。

例 2 では、関数 swap はわたされた 2 つの引数の値を入れ替えます。

仮引数はポインタとして宣言し、呼び出し側ではアドレス演算子を使って、変数 a と b のアドレスをわたしています。

3.2.4 関数 main について

すべてのプログラムには、必ず main という名前の関数があり、実行はこの関数から開始されます。
main は、次に示す引数をもつことができます。

```
main (int argc, char *argv [] , char *envp [] )
```

引数の名前は、他の識別子と重複していなければどのようなものでもかまいません。
argc には、コマンドラインパラメータの個数がわたされます。
argv は、各コマンドラインパラメータを指すポインタの配列です。
envp は、オペレーティングシステムの環境変数を指すポインタの配列です。

```
prog arg1 arg2 arg3
```

プログラム prog を実行するときに、上のようにパラメータを入力すると、main 引数 argc には整数 4 がわたされます。
argv [0] には、プログラム prog という文字列へのポインタがわたされます。
そして、argv [1] から argv [3] までがそれぞれ arg1、arg2、arg3 へのポインタになります。

第４章
構造をもった型

配　列
ポインタ
構造体
共用体
宣言子

本章では、配列、ポインタ、構造体、共用体について説明します。

C言語では、基本データの型の他に、配列や構造体、あるいは共用体といった構造をもった型を定義しています。
これらの型は、基本データ型の組み合わせで構成されています。
ポインタは、構造をもった型ではありませんが、配列との関係が強いので本章で説明します。

4.1 配列

配列とは、同じデータ型の要素の集合です。
文字列も char 型の一次元配列として扱われます。

4.1.1 配列の宣言

配列は宣言によって、配列名、要素の型、要素の数を定義します。

書　式

型指定子　識別子［定数式］［定数式］…；
型指定子　識別子［］；

識別子は配列の名前で、定数式は配列の要素の数を示します。
多次元配列を宣言するときには、定数式を中カッコで囲み、次元の数だけ並べます。
配列の初期化の要素が指定されているときや、ソースプログラムの他の部分で定義済みの配列の参照として宣言するとき、関数の仮引数として宣言するときには、定数式は省略できます。
ただし、中カッコ（［　］）は省略できません。

例 1
```
char name[20] ;
```

例 2
```
int mat [50][50] ;
```

例 3
```
extern char arr[] ;
```

4.1 配 列

例1と例2は、それぞれ一次元、二次元の配列宣言です。
例3は、定義済みの配列の参照宣言ですので、要素の数は省略しています。
配列名は、配列を格納する領域の先頭アドレスを値とする定数として扱われます。

C言語では文字列の終わりを認識するために、文字列の末尾にヌル文字('¥0')をつけ加えます。
そのため、配列を文字定数で初期化したときには、配列の要素の数は文字数より1つ多くなります。
また、文字列定数はその文字列の先頭アドレスを表す定数でもあります。

例1
```
int arry[3] ;
```

例2
```
char carr1[] ="abce";
```

例3
```
char arr2[] = {'a', 'b', 'c', 'd'} ;
```

例1は、要素の数が3個の配列 arry を宣言しています。
例2は、文字列定数で配列を初期化しています。
要素数は、ヌル文字も含めて5になります。
例3は、文字定数を並べて初期化しています。
要素数は4です。

多次元配列は、配列を要素にもつ配列です。
メモリ上では、もっとも右側の添字から先に変化するように割りつけられます。

4.1.2 配列の参照

配列の各要素を参照するには、添字を使って要素を特定します。
各要素に対する操作は、他の変数と同じです。

例 1

```
int i;
          .
          .
          .
          i = arry[x][y];
          .
          .
          .
```

例 2

```
char c;
          .
          .
          .
          c = str[i];
          .
          .
          .
```

例 3

```
if (str[line][pos] == NULL)
        line++;
```

4.1.3 配列の初期化

配列の初期化は、次の書式で行います。

書 式

＝{初期化要素}

配列を初期化するには、宣言に続けて等号（＝）と初期化要素を指定します。
配列はさらに要素を大カッコ（{ }）で囲み、各要素をカンマで区切って並べます。
なお、記憶クラスがautoの配列でも初期化できますが、初期値は単一の式で表されない限り、定数でなければなりません。
配列の初期化で要素数を表す定数式を省略すると、初期化要素の個数が自動的にその配列の要素数の上限になります。

例1

```
int arr[10] = { 1, 2, 3, 4, 5 };
```

例2

```
int arr[] = { 1, 2, 3, 4, 5 };
```

例3

```
int iarr[6][3] = {
        { 1, 2, 3 },     /*  iarr[0]  */
        { 4, 5, 6 },     /*  iarr[1]  */
        { 7, 8, 9 },     /*  iarr[2]  */
        { 1, 1, 1 },     /*  iarr[3]  */
        { 2, 2, 2 },     /*  iarr[4]  */
        { 3, 3, 3 }      /*  iarr[5]  */
};
```

例4

```
int iarr[6][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,
                   1, 1, 1, 2, 2, 2, 3, 3, 3 };
```

例1は、int型の要素を10個もつ配列arrを初期化しています。
初期化要素が足りない部分は自動的に0で初期化されます。
例2の配列では、要素数の上限は5になります。
例3の配列の18個の要素は、注釈で示した順序でメモリに割りつけられます。
初期化もこの順序で行われ、もし、初期化要素が足りなければ0で初期化されます。
例4の初期化は上の例3と同じものです。
また、サイズが明示されている文字配列は、それと同じ数の文字を持つ文字列定数で初期化できます。
その際、ヌル文字は省かれます。

```
char msg[5]="ABCDE";
```

上記の例では、文字配列のサイズは5文字分で、先頭から'A'、'B'、'C'、'D'、'E'が格納されます。

4.2 ポインタ

ポインタは、ポインタ変数などの値の代わりに、その値が格納されているアドレスを示します。
配列もポインタで記述できます。

4.2.1 ポインタの宣言

ポインタの宣言は、ポインタ変数の名前とポインタが指すデータ型を定義します。

書　式

型指定子　識別子；

型指定子には、ポインタを含むすべてのデータ型を指定できます。
識別子は、ポインタ変数の名前です。

例 1
```
int *ptr;
```

例 2
```
int *ptr[5];
```

例 3
```
int (*ptr)[5];
```

例 4
```
int (*ptr)();
```

例1は、int型の対象を指し示すポインタ変数ptrを宣言しています。
例2は、5個のポインタを要素とする配列を宣言しています。
例3は、5個のint型の要素をもつ配列のポインタ変数を宣言しています。
例4は、int型の戻り値をもつ関数へのポインタを宣言しています。
例4の＊とptrを囲むカッコは必要です。
カッコがないと、例3は戻り値がint型へのポインタである関数宣言になります。

4.2.2 ポインタの参照

指し示す対象のデータ型には関係なく、ポインタ変数はint型と同じ大きさです。
XCコンパイラではint型は4バイトですので、ポインタ変数の大きさも4バイトになります。
代入や初期化でポインタ値を0にすると、そのポインタはどのアドレスも指さなくなります。
これをヌルポインタといいます。

ある変数のアドレスをポインタに代入するときには、アドレス演算子（&）を使います。
また、ポインタが指す変数のアドレスの内容を他の変数へ代入するときには、間接演算子（＊）を使います。

int x＝5, y, z, ＊ptr ;

例1

```
ptr = &x;
y = * ptr;
```

例2

```
z = * ptr + 1;
```

4.2 ポインタ

例1は、xの値をyへ代入する操作をポインタを使って間接的に行ったものです。

ポインタ変数ptrにはxのアドレスが代入され、次にptrが指し示すアドレスの内容がyに代入されます。

yの値は5です。

例2は、ポインタの指す内容に対する演算です。

zの値は6になります。

配列の添字を使った要素の参照は、ポインタを使った操作に置き換えることができます。

ポインタに対する演算の結果が意味のあるものとなるのは、ポインタが配列の要素を指しているときです。

プログラム例

```
static  int arr[5] = { 10, 20, 30, 40, 50 };
        int i = 0, x, * parr;
```

例1

```
parr = &arr[i];             /*  parr = arr   */
x = * parr;                 /*  x = arr[i]   */
```

例2

```
x = * parr + 1;             /*  x = arr[i] + 1   */
```

例3

```
x = * (parr + 1);                  /*  x = arr[i + 1]   */
x = parr[i + 1];
```

この例は、ポインタを使った配列操作を示したものです。
例 1 では、まず配列要素のアドレスをポインタ変数に代入しています。
代入後は、ポインタ parr で任意の配列要素を参照できます。
次に parr を指す内容(arr の i 番目の要素)を x に代入します。
x の値は 10 です。
例 2 は、parr が指す内容に整数値 1 を加えたものを、x に代入します。
x の値は 11 です。
この式は演算子の優先順位により、(＊ parr)＋1 として扱われます。
例 3 の式は、すべて同じことを意味します。
最初の式はカッコによって演算子の優先順位を替えています。
parr＋1 の値は、現在 parr が指す要素の次のアドレスです。
x の値は 20 です。
配列は同じデータ型の要素がメモリ上に連続して割りつけられていますので、ポインタが次の要素を参照するためには、要素のデータ型の大きさ（バイト数）だけ進まなければなりません。
この値をアドレスのオフセットといい、ポインタの指すデータ型のバイト数が自動的に使われます。
この列のポインタは int 型を指し示すものなので、ポインタに整数値 1 を加えると 4 バイト先のアドレスを指します。
例では示してありませんが、ポインタ変数と整数値の減算も同じ方法で行われます。

2 つのポインタが同じ配列を指しているとき、ポインタ間の減算ができます。
結果は、差をそのポインタが指すデータ型のサイズ（char であれば 1 バイト、long であれば 4 バイト）で割った符号つき整数値です。
この変換は自動的に行われます。
また、ポインタを関係演算子で比較することもできます。
ポインタの大小関係は、参照する配列要素の順序に従います。

プログラム例

```
static  int arr[10];
        int x, * px, * py;
px = &arr[5];
py = &arr[2];
```

4.2 ポインタ

例1

```
x = px - py;
```

例2

```
if (px > py) {
        func();
}
```

例1は、ポインタ間の減算です。
xの値は整数値3です。
例2は、ポインタの比較です。
結果は真になるので、if文の本体は実行されます。

ポインタを要素にもつ配列を使って、長さの違う文字列を配列の1要素のように扱うことができます。
次の例に示す関数printmsgは、引数によってそれぞれのメッセージが出力されます。
static変数がポインタ配列の宣言と初期化です。

プログラム例

```
viod ptinrmsg(msgno)
int msgno;
{
        static char * msg[] = {
                "Elliegal",
                "Message I.",
                "Message II.",
                "Message III."
        };

        fprintf(stderr,
                ((msgno < 1) || (msgno > 3))
                        ? msg[0] : msg[msgno]);
};
```

4.2.3 ポインタの初期化

ポインタの初期化は、次の書式で行います。

書　式

＝ 式

ポインタを初期化するには、宣言に続けて符号（＝）と初期値となる式を指定します。

例1

```
int *ptr=0;
```

例2

```
char c='a';
char *cptr=&c;
```

例1と例2は、ポインタの初期化です。
例2では、変数cのアドレスをポインタの初期化要素に指定しています。

4.2.4 ポインタによる引数の受けわたし

第3章でもふれましたが、C言語では関数呼び出しのときの引数はすべて値（正確には値の一時的なコピー）でわたします。
このため、呼び出された関数内でその値を直接変更できません。
そこで、仮引数をポインタとして宣言し、呼び出す引数のアドレスをわたすことができます。

プログラム例

```
main()
{
        char st1[10];
        static char st2[] = "string";
        .
        .
        .
        copy(st1, st2); '
        .
        .
        .
}

char * copy(s1, s2);
char * s1, * s2;
{
        while( * s1++ = * s2++)
                ;
}
```

上の例の関数copyは、文字例のコピーを行います。
配列名はその配列の先頭アドレスを表すので、copyを呼び出す側では引数のアドレスをわたしていることになります。
関数copyの本体は、少しわかりづらい表現になっていますが、これは次のものと同じです。

```
int i=0;
while ((s1[i] =s2[i]) !='¥0')
                i++;
```

演算子の結合規則により＊s1＋＋は＊(s1＋＋)の意味です。
仮引数には配列の先頭アドレスがわたされています。
まず、st2[0] の値 's' がst1[0] に代入されます。

4.2　ポインタ

その後、それぞれのポインタがインクリメントされ、次の文字が代入されます。

文字列の最後のヌル文字('¥0')が代入されたとき式の評価は偽となり、while文字は終了します。

関数名は、配列名と同じくその関数の先頭アドレスを表します。

関数の引数に他の関数をわたすときは、関数へのポインタを使います。

次の例は関数へのポインタの使いかたを示しています。

関数funcの引数fpが、関数へのポインタとして宣言されています。

プログラム例

```
main()
{
        int x, y, z;
        int suba();
        .
        .
        .
        func(x, y, z, suba);
        .
        .
        .
}

func(a, b, c, fp)
int a, b, c;
int (* fp)();
{
        .
        .
        .
        (* fp)(c);
        .
        .
        .
}

suba(ch)
int ch;
{
        .
        .
        .
        return;
}
```

4.2.5 void へのポインタ

void へのポインタを表す型は、void * と表現されます。
このポインタ型 void * は、任意のポインタが情報を失うことなくキャストでき、また逆に元のポインタへとキャストしてもその内容は変わりません。
型の違うポインタを混在させる場合、必ず明示的な変換をしなければなりません。
しかし、void へのポインタは、そのまま他のどんな型のポインタとも混在させることが可能です。

4.3 構造体

構造体は、型の異なる複数の変数の集合を１つのデータ単位として扱うものです。
構造体を構成する変数をメンバと呼びます。

4.3.1 構造体の宣言

構造体の宣言は、次の書式で行います。

書　式

struct［タグ名］｛メンバ宣言リスト｝［識別子］；

［と］の中は、省略できることを表します。
タグ名は、宣言する構造体の型名です。
メンバ宣言リストは、構造体のメンバである変数の宣言です。
メンバには、構造体を含むすべてのデータ型を指定でき、異なるデータ型の変数を指定できます。
識別子は、宣言する構造体をもつ変数名を定義します。

例1

```
struct {
        int p, q;
        double x;
} strver;
```

例2

```
struct mrec {
        int month, day;
        char name[20];
        int id;
};
```

例3

```
struct mrec mlist[MAX], staff;
```

例4

```
struct temp {
        char c;
        long lx, ly;
        int * ptr;
        struct trmp * next;
} xtemp;
```

例1は、構造体strvarの宣言です。

タグ名は省略しています。

例2は、型宣言です。

型名mrecを定義しています。

例3は、定義済みのタグ名を使った構造体の宣言です。

mlistは、構造体を要素にもつ配列になります。

例4のメンバnextは、宣言している構造体自身へのポインタです。

構造体は、それを1つの単位として、基本データ型の変数と同じように扱うことができます。

ここでは、次の項目について、例をあげて説明します。

- 構造体の配列
- 構造体の代入
- 構造体と関数

構造体の配列

配列の要素に構造体を指定して宣言できます。

要素の大きさ（オフセット）は、メンバのデータ型のサイズを合計したものです。

プログラム例

```
struct record {
        int id;
        char * name;
        int sex;
        int w_time;
};
struct record s_list[MAX];
struct record * lstp;
```

例 1

```
s_list[10].id = 1001;
```

例 2

```
lstp = s_list;              /*  listp = &s_list[0]   */
(++lstp)->name = "WINK-Aida";
```

例 3

```
int i;
for ( i = 0; i < (MAX + 1); i++) {
        printf("ID = %4d %20s %c %4dH¥n",
                s_list[i].id, s_list[i].name,
                s_list[i].sex == 1 ? 'M' : 'F',
                s_list[i].w_time);
}
```

この例では、まず型宣言により型名 record の構造体を宣言しています（この record 型は、この後の例にも使います)。

次に、構造体を要素にもつ配列変数と構造体へのポインタを宣言しています。

例 1 は、11 番目の要素のメンバ id への代入です。

例 2 の文字列定数は、2 番目の要素のメンバ name に代入されます。

例 3 は、配列要素の一覧表を出力します。

構造体の代入

2つの構造体が同じメンバ構成のとき（各メンバのデータ型と宣言の順序が同一）には、構造体のメンバの内容を、まとめてもう1つの構造体に代入できます。

メンバ構成が異なる構造体間の代入では、結果の値は保証されません。

プログラム例

```
struct record rec1 = {
        2736,               /*  id   */
        "Miyazawa",         /*  name   */
        2,                  /*  sex   */
        42                  /*  w_time   */
};
        .
        .
        .
func()
{
        static struct record rec = rec1;
        .
        .
        .
}
```

この例では、関数 func の中で構造体 rec の初期化要素としてブロックの外側で初期化した rec1 を指定しています。

構造体と関数

構造体そのものを関数の引数として指定すると、関数にはその構造体のコピーがわたされます（値による呼び出し）。

また、構造体へのポインタをわたすこともできます。

関数の戻り値にも、構造体そのものと構造体へのポインタの両方を指定できます。

プログラム例

```
struct record rec;
main()
{
        struct record data, func4(), * func3(), * p;
        .
        .
        .
        func1(&data);
        .
        .
        .
        data = func4();
        .
        .
        .
        func2(data);
        .
        .
        .
        p = func3();
        .
        .
        .
}

func1(ptr)
struct record * ptr;
{
        int i;
        .
        .
        .
}

struct record func4()
{
        .
        .
        .
        return(rec);
}

func2(dt)
struct record dt;
{
        int i;
        .
        .
        .
}

struct record func3()
{
        .
        .
        .
        return(&rec);
}
```

この例の関数 main の中では、4個の関数が呼び出されています。

func1は引数が構造体へのポインタ、func2は構造体そのものです。
また、func3は構造体へのポインタを戻り値として返し、func4は構造体そのものを返します。

ビットフィールドメンバ

ここまで説明したものの他に、構造体メンバの特別な型としてビットフィールドの宣言があります。
ビットフィールドメンバの宣言の書式は、次の通りです。

書　式

unsigned ［識別子］：定数式

［と］で囲まれたものは、省略が可能です。
識別子は、メンバ名（フィールド名）です。
メンバの型は、必ずunsigned int型です。
定数式は、メモリに割りつけられるビット数を表し、正整数で1からint型のサイズ（ビット数）までを指定します。
XCコンパイラのint型は、32ビット（4バイト）なので、定数式の値は1から32までになります。
識別子は、省略できます。
省略するとメモリ割りつけは行われますが、その部分の参照はできません。
XCコンパイラでは、32ビット（4バイト）を1ワードとして扱います。
ビットフィールドメンバは、ワードの先頭から宣言した順序で割りつけられ、メンバがワード境界にまたがることはありません。
自動的に次のワードの先頭から割りつけられます。
ワード境界の調整を強制的に行うには、調整するメンバの前に、メンバ名を省略した定数式の値が0のメンバを宣言します。
ビットフィールドは、1ビット単位でメモリに割りつけられているので、ビットフィールドの配列やビットフィールドを指すポインタは宣言できません。

例1

```
struct {
        unsigned a: 2;
        unsigned b: 15;
        unsigned c: 10;
        unsigned d: 8;
} x;
```

例2

```
struct {
        unsigned a: 2;
        unsigned b: 15;
        unsigned  : 0;
        unsigned c: 10;
        unsigned d: 8;
} y;
```

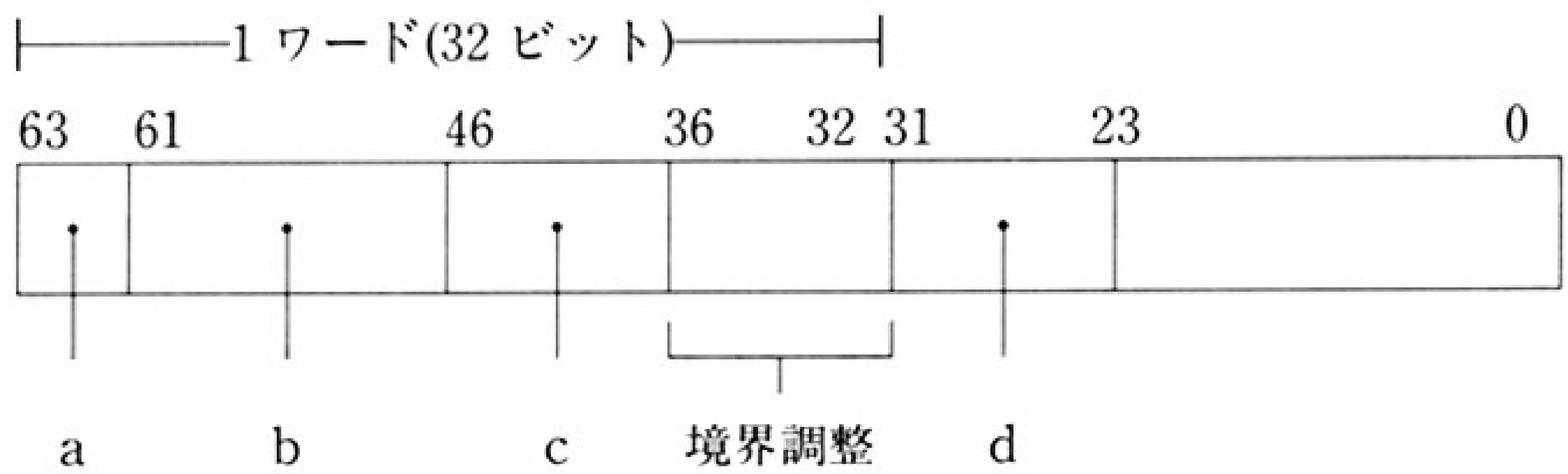
例3

```
#define ON  1
#define OFF 0

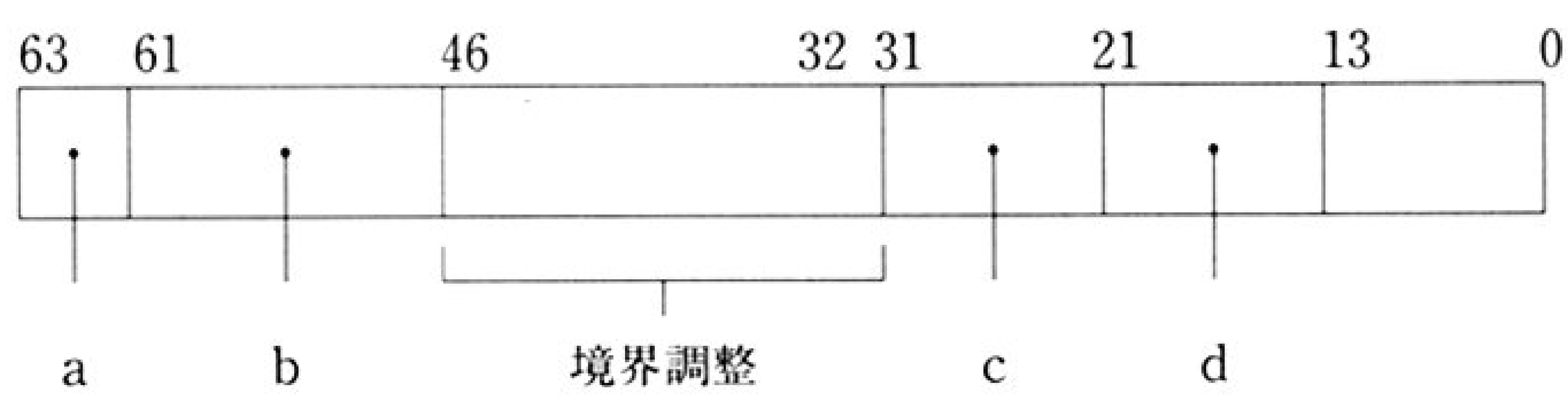
struct {
        unsigned bit_0: 1;
        unsigned bit_1: 1;
        unsigned bit_2: 1;
        unsigned bit_3: 1;
} flag;

flag.bit_0 = flag.bit_3 = ON;
flag.bit_1 = flag.bit_2 = OFF;
```

例1の構造体xは、メモリ上に次のように割りつけられます。



例2は、強制的なワード境界の調整です。
メモリ上に次のように割りつけられます。



例3は、ビットフィールドメンバの参照です。

4.3.2 構造体の参照

構造体のメンバの参照は、次の書式で行います。

書 式

構造体変数名.メンバ名

構造体へのポインタ変数名－＞メンバ名

プログラム例

```
struct sample {
        int x;
        char c;
};
struct sample data1;
struct sample * ptr;
```

例 1

```
data1.x = 152;
```

例 2

```
ptr = &data1;
ptr->c = 'A';            /*  (* ptr).c = 'A'   */
```

例 1 は、メンバ演算子 “．” を使ったメンバ参照です。

例 2 は、構造体へのポインタを使ったものです。

注釈文内の記述を簡略化して表現しています。

4.3.3 構造体の初期化

構造体の初期化は、次の書式で行います。
なお、記憶クラスがautoの構造体でも初期化できますが、初期値は単一の式で表されない限り、定数でなければなりません。

書　式

＝{初期化要素}

プログラム例

```
static struct sample {
        int x, y;
        char ch[2];
} svar = {
        5,
        6,
        {'A','B'}
};
```

例では、xとyをそれぞれ5と6に、配列ch［0］からch［1］をそれぞれ'A'と'B'に初期化しています。

4.4 共用体

共用体は、1つの領域に格納された値を、異なるデータ型や異なる識別子で共用するものです。

4.4.1 共用体の宣言

共用体の宣言は、次の書式で行います。

書　式

union［タグ名］｛メンバ宣言リスト｝［識別子］；

［と］で囲まれたものは、省略が可能です。
書式は、構造体の宣言と同じです。
ただし、ビットフィールドは指定できません。

例1

```
union {
        int s;
        unsigned u;
        float f;
} var;
```

例2

```
union ut {
        char * names;
        struct {
                char name[20];
                int id;
        } s_names;
};
```

例 3

```
union ut sample, temp;
```

4.4.2 共用体の参照

共用体は、メンバ間のオフセットが 0 の構造体と考えられます。
したがって、メンバの参照やその他の事項についても、ビットフィールドを除いて、すべて構造体と同様に扱うことができます。
共用体の大きさは、宣言したメンバのもっとも大きいデータ型サイズと同じです。
共用体を使うと、同一メモリ上に異なるデータ型の値を保存できます。
保存している値の型は、最後に代入などによって参照したメンバのデータ型です。
ただし、現在共用体がどのデータ型の値を保持しているか、という情報をコンパイラは保存していません。
もし、保存している値のデータ型と異なる型のメンバで参照すると、結果は保証されません。

プログラム例

```
union tag {
        short    sh;
        int      in;
        double   db;
        unsigned un;
        char     ch[10];
};
```

例 1

```
union tag var;
var.sh = 123;
var.in = 48679;
var.db = 79734.01;
var.un = 0xF7C1;
var.ch = "pb-tech";
```

例 2

```
union tag var, * ptr;
ptr = &var;
ptr->in = 4080;
```

例 3

```
double x;
union tag var;
var.ch = "Tomono";
x = var.db;      /*  unknown  */
```

例は、共用体メンバの参照です。

例 1 と例 2 は、それぞれ演算子“.”と“−>”を使ったものです。

例 3 の x の値は確定できません。

共用体のデータ型の管理はプログラマの責任なので、コーディング上の工夫が必要になります。

プログラム例

```
enum sex  {male, female};
enum flag {staffA, staffB, other};

struct person {
        int id;
        char * name;
        enum sex s_sex;
        short birth;
        enum flag s_flag;
        union {
                short w_time;
                short w_day;
                char * memo;
        } sc;
        int pay;
} staff[MAX];
int i, cost_a, cost_b;
        .
        .
        .
switch (s_flag) {
        case staffA: staff[i].pay = staff[i].sc.w_time * cost_a;
                     break;
        case staffB: staff[i].pay = staff[i].sc.w_day * cost_b;
                     break;
        case other : staff[i].pay = 0;
                     staff[i].sc.memo = "== none ==";
                     break;
```

この例では、共用体scのどのメンバを参照するかを判断するフラグとして、変数s_flgを使っています。

4.4.3 共用体の初期化

共用体の初期化は、次の書式で行います。
共用体の初期値式は、同じ型の単一の式か、あるいは、その共用体の最初のメンバに対する大カッコ{}つきの式のどちらかです。
なお、記憶クラスがautoの共用体でも初期化できますが、初期値は単一の式で表されない限り、定数でなければなりません。

書　式

＝{初期化要素}

プログラム例

```
static union sample {
        int iarr[3];
        float x, y;
} uvar = {
        { 10, 20, 30 }
};
```

初期化は、最初に宣言されているメンバiarrについてのみ行います。

4.5 宣言子

ポインタ宣言や配列宣言、関数宣言のとき、宣言する識別子を、特に宣言子といいます。
このとき、中カッコ（［　］）やアスタリスク（*）、カッコは、修飾子といいます。
たとえば、name という char 型の 20 個の配列を宣言するときは、

```
char name［20］；
```

とします。
この配列宣言で、宣言子 name は配列修飾子［20］で修飾されています。
一般に、宣言は次の書式で行います。
記号は、すべて書式の一部です。

識別子
宣言子［　］
宣言子［定数式］
*宣言子
宣言子（　）
宣言子（引数型リスト）
(宣言子)

この節では、宣言子の作成方法と、その読みかたについて説明します。

4.5.1 複合宣言子

複合宣言子とは、複数の修飾子によって修飾されている宣言子のことです。
複数の修飾は、宣言に応じていろいろな組み合わせが考えられますが、次のような宣言はできません。

- 関数の配列
- 配列や関数を戻り値として返す関数

複合宣言子は、次の規則に従って読んでいきます。

1. 配列宣言と関数宣言は、ポインタ宣言より優先度が高い
2. 配列宣言と関数宣言は同じ優先度で、左から右に読む
3. 優先度は、カッコをつけて変えることができる

実際には、複数のカッコ（優先度を変えるためのもの）のもっとも内側に宣言子（識別子）があるので、そこから解釈を始めます。
宣言子から右方向に左カッコ（関数修飾子）か中カッコ（配列修飾子）を探し、次に左方向にアスタリスク（ポインタ修飾子）を探します。
このとき、優先度を変えるカッコがあれば、それに従います。
次の例で説明します。

```
char (*(* name)( )) [20] ;
```

それでは、読みかたに従ってみていきます。
一番内側にあるのはnameですので、これは識別子nameの宣言です。
nameはポインタで、値を返す関数を指します。
関数はポインタを返し、このポインタは20個の要素をもつ配列を指しています。
その配列の要素はchar型です。

例1

```
/* name は、char 型の値を指すポインタの配列 */
char *name[20] ;
```

例2

```
/* name は、char 型の値の配列を指すポインタ */
char (*name)[20] ;
```

例 3

```
／＊ code は、int 型の値を指すポインタを返す関数＊／
int ＊code(int, int) ;
```

例 4

```
／＊ code は、int 型の値を返す関数を指すポインタ＊／
int (＊code)(int, int) ;
```

例 3 と例 4 の関数の引数は、抽象宣言子です。
抽象宣言子については、この後で説明しています。
複合宣言子は、不用意に使うとわかりにくいものになるので、注意してください。

4.5.2 抽象宣言子

抽象宣言子とは、宣言の特別な形式で、修飾子だけで構成される宣言子です。
ポインタ修飾子は、必ず宣言子の左に置きます。
また、配列修飾子と関数修飾子は、必ず宣言子の右に置きます。
このため、コンパイラは抽象宣言子を正しく解釈できます。
また、抽象宣言子は、複数の修飾子で修飾できます（複合抽象宣言子）。
このときの読みかたの規則は、複合宣言子と同じです。

例

```
／＊ int 型の値の配列を指すポインタを返す関数＊／
int (＊code(int (＊)[10]));
```

例の関数は、引数が抽象宣言子になっています。
int (＊) のカッコは必ずつけます。
カッコがないと、引数は int 型の値を指すポインタの配列と解釈されます。

第5章

プリプロセッサ

マクロ定義
ファイルのとり込み
条件つきコンパイル
行制御

本章では、プリプロセッサとプリプロサッサ命令について説明します。

プリプロセッサとは、XC コンパイラがソースファイルをコンパイルする前に、マクロ定義、条件コンパイル、または、他のファイルからのとり込みなどの前処理を行うプログラムです。
これらの制御を行うのがプリプロセッサ命令です。
XC コンパイラのプリプロセッサ命令を以下に示します。

# define	# elif
# else	# endif
# if	# ifdef
# ifndef	# include
# line	# undef

プリプロセッサ命令は、上記のようにシャープ記号(#)を先頭につけて記述します。
以下に示すように、'#'は行の先頭からでなくてもよく、また'#'とトークンの間には、タブやスペースが存在しても構いません。

```
  # define MAX   56
   # ifdef ENCODE
```

文末は、セミコロン（;）ではなく改行です。
また、プリプロセッサの引数に以下のようなオペレータを使用することで、トークンの加工を行うことができます。

#トークン	：トークンの文字列化
トークン##トークン	：トークンの連結

例えば、

```
 # define sample (str)        # str "コンパイラ"
```

と定義されたマクロは次のように展開されます。

```
 sample (XC)➡"XC" "コンパイラ"➡"XC コンパイラ"
```

また、

```
 # define sample2 (a, b)        a ## b
```

と定義されたマクロは次のように展開されます。

```
 sample2 (ABC, DEF)➡ ABCDEF
```

プリプロセッサ命令文は、プログラム中のどこにでも記述できます。
しかし、その命令が有効になるのは、その行以降です。

5.1 マクロ定義

たとえば、定数などを意味のある名前に置き換えておくと、プログラムが見やすくなり、メンテナンスも楽になります。
これをマクロ定義といいます。
マクロ定義は、関数と機能的に似ていますが、次の点で異なります。

- マクロ定義では複文を記述できない。
- マクロ定義された識別子は、プリプロセッサによりすべて置き換えられるので、プログラムが大きくなる。

あまり複雑でない処理は、マクロ定義にしておくことをおすすめします。
それにより、関数を呼び出すオーバーヘッドが避けられるからです。
なお、二重のマクロ定義はエラーとはならず、警告が出力されます。

5.1.1 # define

defineは、マクロを定義します。

書　式

　# define　識別子　[文字列]
　# define　識別子(引数リスト)　[文字列]

[と]で囲まれたものは、省略が可能です。
上の書式の場合、# defineで定義した識別子をソースプログラム中でみつけると、指定した文字列に置き換えます。
文字列を省略すると、定義した識別子をソースプログラムから削除します。
しかし、マクロ定義は解除されません。
識別子と文字列は、1文字以上の空白文字で区切らなければなりません。
2つ目の書式は、文字列が引数をもつ場合で、引数リストは文字列中で使われている仮引数をカンマ(,)で区切って、並べたものです。
defineで定義した識別子をソースプログラム中でみつけると、文字列中の仮引数を実引数に置き換えたうえで、識別子を文字列に置き換えます。
なお、以下のようにあらかじめ定義されているマクロが存在するので注意し

て下さい。

__LINE__：ソースファイル中の行番号を表す10進定数
__FILE__：ソースファイル名を表す文字列定数
__STDC__：ANSI規格版番号（準拠している場合は1）
__TIME__：コンパイルの日付を表す文字列定数（"Mmm dd yyyy"）
__DATE__：コンパイルの時間を表す文字列定数（"hh : mm : ss"）

例1
```
#define  PI  3.141592
```

例2
```
#define  ERRMSG "数字は半角で入力してください"
```

例3
```
#define  MAX (X, Y)  ((X)>(Y)?(X):(Y))
```

例4
```
#define  UNCHAR  unsigned char
```

例5
```
#define  SQR (a)  (a)*(a)
```

例6
```
#define  SQR (a)  a * a
```

例1では、定数3.141592をPIとして定義しています。
例2では、エラーメッセージを定義しています。
2行にわたる文字列は、円記号（¥）でつなぎます。
例3で定義したマクロMAXを式1で使うと、式2のように置き換えられます。

式1　　max=MAX (a+b, c+d)
式2　　max=((a+b)>(c+b)?(a+d):(c+b))

#defineで定義する場合は、式の評価順序を明確にするためにカッコ（　）を注意深く使ってください。

例4は、予約語を再定義しています。
これにより

UNCHAR c;

という宣言が可能になります。

例5、例6のマクロは、文字列の引数aのカッコの有無だけが異なります。これらのマクロを次の式3で使うと、それぞれ式4、式5のように置き換えられます。

式3　　sgr=SQR (x+1)
式4　　sgr=(x+1)*(x+1)
式5　　sgr=x+1 * x+1

5.1.2 #undef

#undefは、#defineによるマクロ定義をとり消します。
#defineとの組み合わせにより、マクロ定義の有効範囲を指定します。

書　式

#undef　識別子

プログラム例

```
#define MAXLINE 255
#define sub(x, y)   ((x)-(y))
          .
          .
          .
#undef sub
#undef MAXLINE
```

5.2 ファイルのとり込み

プログラムの生産性の向上や汎用化のために、頻繁に使うルーチンや宣言をあらかじめ別のテキストファイルとして作成しておくことができます。
このテキストファイルを、インクルードファイルといいます。
XCコンパイラには、いくつかのインクルードファイルが用意されています。
これについては、「Cライブラリマニュアル」を参照してください。
プログラム作成時にインクルードファイルをとり込み、位置を指定するだけで、プリプロセッサによりその位置にとり込まれます。

5.2.1 # include

include は、インクルードファイルをとり込む位置を指定し、その位置にインクルードファイルの内容が挿入されます。
インクルードファイル中に# include 命令を記述することもできます。

書　式

```
# include　"ファイル名"
# include　<ファイル名>
```

ファイルをダブルクォーテーション(" ")で囲むと、プリプロセッサはカレントディレクトリ、コンパイラオプションで指定したディレクトリ、環境変数で定義したディレクトリの順で指定したファイルを探します。
ファイル名を角カッコ（< >）で囲むと、コンパイルオプションで指定したディレクトリ、環境変数で定義したディレクトリの順に探します。

例1
```
#include    <stdio.h>
```

例2
```
#include    "ctype.h"
```

例3
```
#include    "A:¥sys¥mydir¥deffnc.h"
```

5.3 条件つきコンパイル

C言語では、条件によって部分的にコンパイルを抑止することができます。
これを条件つきコンパイルといいます。
プリプロセッサは、コンパイルしない部分をソースファイルから取り除きます。

5.3.1 #if、#elif、#else、#endif

#if～#elif～#else～#endif は、コンパイルを抑止する条件を設定し、コンパイルしない部分を指定します。

書　式

```
#if [defined]   定数式
[#elif          定数式
            テキスト]
[#else
            テキスト]
#endif
```

[と] で囲まれたものは、省略が可能です。
#if は、必ず#endif で終了しなければなりません。
#elif は、#if と#endif の間に何回でも記述できます。
#else は、#endif の直前に 1 回だけ記述できます。
定数式に整数の sizeof、型キャスト、列挙型定数を含むことはできません。
defined という演算子を使うことができます。
defined に続いて記述されたトークンが、それまでにプリプロセッサにより定義されていたら 1L（真）に、そうでなければ 0L（偽）に変換されます。
テキストは、コンパイラやプリプロセッサが理解できる命令、式、文、関数などで、1 行以上記述することができます。
プリプロセッサは、評価結果が真（0 以外）の定数式がみつかるまで、#if または#elif の定数式を順に評価します。
真の定数式がみつかると、そこから次のシャープ記号(#)の間にあるテキストの選択をします。

5.3 条件つきコンパイル

真の定数式がない場合は、#else と#endif の間のテキストが選択されます。真の定数式がなく、かつ#else が省略されている場合は、#if から#endif までのすべてのテキストが取り除かれます。
なお、#if、#elif、#else、#endif 文において、#else の後ろにトークンが存在してはいけません。

例1

```
#if     defined(VAR1)
          func1();
#elif   defined(VAR2)
          func2();
#else
          errors();
#endif
```

例2

```
#if     MAXVAR > 10
          #define FLAG 1
          #if SETFLG == 1
                #define  VARS 3
          #else
                #define  VARS 2
          #endif
#else
          #define FLAG 0
          #if SETFLG == 1
                #define  VARS 2
          #else
                #define  VARS 4
          #endif
#endif
```

例3

```
#if     SETVAR == 1
          #define LIMIT 1000
#elif   SETVAR == 0
          #define LIMIT 500
#elif   SETVAR  > 2
          #define LIMIT 3000
#else
          errors();
#endif
```

例1では、#if と#endif が3つの関数呼び出しのうちの1つをコンパイルします。

識別子 VAR1 が定義済みであれば、func1 に対する関数呼び出しをコンパイルします。

識別子 VAR2 が定義済みであれば、func2 に対する関数呼び出しをコンパイルします。

どちらの識別子も定義されていなければ、#else 命令の次の errors の呼び出しをコンパイルします。

例2では、あらかじめ#define 命令で MAXVAR に定義してあるものとします。

2組のネストした#if、#else、#endif を使っています。

1組目の#if は、式 MAXVAR>10 が真のときだけ処理します。

その他の場合には、2組目の#if を処理します。

例3では、あらかじめ SETVAR に定数を定義してあるものとします。

SETVAR の値にもとづいて、4個のブロックから1つを選ぶために#elif と#else を使います。

定数 LIMIT は、SETVAR の定数値の定義によって 500、1000、3000 のどれかにセットされます。

SETVAR を定義しないと、errors ( ); をコンパイルし LIMIT は定義しません。

5.3.2 #ifdef、#ifndef

#ifdef は、#if の定数式が defined（識別子）の場合と同じです。

#ifndef は、#ifdef と反対の条件を調べます。

書　式

```
#ifdef   識別子
#ifndef  識別子
```

#ifdef は、識別子がすでに#define で定義されていれば真になります。

#ifndef は、識別子が定義されていないか、#undef でとり消されているときに真になります。

5.4 行制御

コンパイラには、コンパイルエラーが発生した行番号とファイル名を内部的に記憶する機能があります。
プリプロセッサには、この行番号とファイル名を変更する機能が用意されています。

5.4.1 #line

#lineは、コンパイラの内部に記憶されている行番号とファイル名を変更します。
変更後は、以前の行番号とファイル名は無視され、新しい行番号とファイルで処理が続けられます。

書　式

#line 定数　[”ファイル名”]

定数は整数値です。
［と］で囲まれたものは、省略が可能です。
ファイル名は、必ずダブルクォーテーション（”　”）で囲まなければなりません。
現在の行番号とファイルは、あらかじめ定義されているマクロ__LINE__と__FILE__で参照できます。

例

```
#line　500　”myfunc. c”
```

付　録

構文の要約

トークン

式

宣　言

文

定　義

プリプロセッサ命令

構文の解釈は、次の例を参照してください。

例

A：：=B	Aとは、Bである
B：：=C｜D｜E	Bとは、CかDかEのどれかである

1 トークン

トークン：：＝

予約語｜識別子｜定数｜文字リテラル｜演算子｜区切子

1.1 予約語

予約語：：＝

auto | break | case | char | const | continue | default | do | double | else | enum | extern | float | for | goto | if | int | long | register | return | short | signed | sizeof | static | struct | switch | typedef | union | unsigned | void | volatile | while

1.2 識別子

識別子：：＝

英字｜下線｜識別子　英字｜識別子　下線｜識別子　数字

英字：：＝

a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

下線：：＝

_

数字：：＝

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

1.3 定数

定数：：＝

整数定数｜long 型定数｜浮動小数点定数｜enum 型定数｜文字定数

整数定数：：＝

0｜10 進定数｜8 進定数｜16 進定数

10 進定数：：＝

0を含まない数字 | 10進定数　数字

0を含まない数字：：=

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

8進定数：：=

0　8進数字 | 8進定数　8進数字

8進数字：：=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

16進定数：：=

0x　16進数字 | 0x　16進数字 | 16進定数　16進数字

16進数字：：=

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

long型定数：：=

整数定数L | 整数定数l

浮動小数点定数：：=

小数定数 | 小数定数　指数 | 数字列　指数

数字列：：=

数字 | 数字列　数字

小数定数：：=

数字 | 数字列. | 数字列.　数字列 | .　数字列

指数：：=

e符号　数字列 | E　符号　数字列 | e　数字列 | E　数字列

符号：：=

+ | −

1　トークン

enum 型定数：：＝

識別子

文字定数：：＝

'文字'

文字：：＝

英字｜数字｜エスケープシーケンス｜シングルクォーテーション（'）、円記号（¥）を除いた特殊文字

エスケープシーケンス：：＝

¥n｜¥t｜¥v｜¥b｜¥r｜¥f｜¥'｜¥"｜¥¥｜¥0｜¥000｜¥xhh｜¥Xhh｜¥a

・000 は 8 進数字、hh は 16 進数字を表す

特殊文字：：＝

句読文字｜空白文字

句読文字：：＝

，｜.｜；｜：｜?｜'｜"｜（｜）｜［｜］｜｛｜｝｜<｜>｜!｜
｜｜／｜¥｜￣｜_｜#｜%｜&｜^｜＊｜－｜＝｜＋

空白文字：：＝

｜¥t｜¥v｜¥r｜¥n｜¥f｜¥a

文字列定数：：＝

" "｜"文字列"

文字列：：＝

文字｜数字｜エスケープシーケンス｜半角カナ文字｜ダブルクォーテーション（"）を除いた特殊文字｜2 バイト系文字｜文字列　英字｜文字列　数字｜文字列　エスケープシーケンス｜文字列　半角カナ文字｜文字列　ダブルクォーテーション（"）を除いた特殊文字｜文字列 2 バイト系文字

半角カナ文字：：＝

ｱ｜ｲ｜ｳ｜ｴ｜ｵ｜ｶ｜ｷ｜ｸ｜ｹ｜ｺ｜ｻ｜ｼ｜ｽ｜ｾ｜ｿ｜
ﾀ｜ﾁ｜ﾂ｜ﾃ｜ﾄ｜ﾅ｜ﾆ｜ﾇ｜ﾈ｜ﾉ｜ﾊ｜ﾋ｜ﾌ｜ﾍ｜ﾎ｜
ﾏ｜ﾐ｜ﾑ｜ﾒ｜ﾓ｜ﾔ｜ﾕ｜ﾖ｜ﾗ｜ﾘ｜ﾙ｜ﾚ｜ﾛ｜ﾜ｜ｦ｜
ﾝ｜ｧ｜ｨ｜ｩ｜ｪ｜ｫ｜ｬ｜ｭ｜ｮ｜ｰ｜ﾞ｜ﾟ｜､｜｡｜･｜
｢｜｣｜

2バイト系文字：：＝

0X8140 から 0XEBDD の文字

1.4 演算子

演算子：：＝

!｜‾｜+｜−｜＊｜／｜%｜<<｜>>｜<｜<=｜>｜>=
｜==｜!=｜&｜|｜^｜&&｜‖｜,｜?:｜++｜−−｜=
｜+=｜−=｜＊=｜／=｜%=｜<<=｜>>=｜&=｜|=｜
^=｜[ ]｜( )｜.｜−>｜sizeof｜#｜##｜defined

1.5 区切記号

区切記号：：＝

[｜]｜(｜)｜{｜}｜＊｜,｜:｜=｜;｜#

1.6 注釈

注釈：：＝

／＊『＊／』の組み合わせを含まない文字列　＊／

2　式

式：：＝

識別子｜文字列｜式（式リスト）｜式（　）｜式［式］｜式．識別子｜式－＞識別子｜単項式｜二項式｜三項式｜代入式｜（式）｜（型名）式｜定数式

式リスト：：＝

式｜式リスト，式

単項式：：＝

単項演算子　式｜sizeof（式）

単項演算子：：＝

－｜¯｜!｜＊｜&

二項式：：＝

式　二項演算子　式

二項演算子：：＝

＊｜／｜%｜＋｜－｜＜＜｜＞＞｜＜｜＞｜＜＝｜＞＝｜＝＝｜!=｜&｜｜｜^｜&&｜‖｜,

三項式：：＝

式？式：式

代入式：：＝

左辺値＋＋｜左辺値－－｜＋＋左辺値｜－－左辺値｜左辺値　代入演算子　式

左辺値：：＝

識別子｜式［式］｜式．式｜式－＞式｜＊式｜(型名）式｜(左辺値)

代入演算子：：＝

＝｜＋＝｜－＝｜＊＝｜／＝｜＜＜＝｜＞＞＝｜&＝｜｜＝｜^＝

定数式：：＝

識別子｜定数｜(型名）定数式｜単項式｜二項式｜三項式｜(定数式)

・型名は、「付録3　宣言」を参照

3 宣　言

宣言：：＝

記憶クラス指定子　型指定子　宣言子リスト；｜記憶クラス指定子　宣言子リスト；｜型指定子　宣言子リスト；型指定子；｜typedef 型指定子　宣言リスト；

記憶クラス指定子：：＝

auto｜extern｜register｜static

型指定子：：＝

char｜double｜float｜int｜long　int｜short　int｜short｜signed char｜signed int｜signed long int｜signed long｜signed short int ｜signed　short｜unsigned｜unsigned　char｜unsigned　int｜unsigned　long　int｜unsigned　long｜unsigned　short　int｜unsigned short｜enum 指定子｜構造体指定子｜共用体指定子｜typedef 名

enum 指定子：：＝

enum　タグ｛enum リスト｝｜enum｛enum リスト｝｜enum タグ

タグ：：＝

識別子

enum リスト：：＝

列挙子｜enum リスト，列挙子

列挙子：：＝

識別子｜識別子＝定数式

構造体指定子：：＝

struct タグ｛メンバ宣言リスト｝｜struct｛メンバ宣言リスト｝｜struct タグ

メンバ宣言リスト：：＝

メンバ宣言｜メンバ宣言リスト　メンバ宣言

3 宣言

メンバ宣言：：＝

型指定子　宣言子リスト；|型指定子　識別子　：定数式；|
型指定子　：定数式；

宣言子リスト：：＝

宣言子|宣言子＝初期化要素|宣言子リスト，宣言子

宣言子：：＝

識別子|宣言子［定数式］|宣言子［　］|＊宣言子|宣言子（引数型リスト）|宣言子（　）|（宣言子）

引数型リスト：：＝

型名|引数型リスト，型名|引数型リスト，|void|void ＊

型名：：＝

型指定子|型指定子　抽象宣言子

抽象宣言子：：＝

＊|［　］|（引数型リスト）|＊抽象宣言子|抽象宣言子＊|
［定数式］抽象宣言子|［　］抽象宣言子|抽象宣言子［定数式］
|抽象宣言子［　］|抽象宣言子（引数型リスト）|抽象宣言子（　）|（抽象宣言子）

初期化要素：：＝

式|｛初期化要素リスト｝

初期化要素リスト：：＝

初期化要素|初期化要素リスト，初期化要素

共用体指定子：：＝

union タグ｛メンバ宣言リスト｝|union｛メンバ宣言リスト｝|
union タグ

typedef 名：：＝

識別子

4 文

文：：＝

式；｜break；｜case 定数式：文｜continue；｜default：文｜do 文 while（式）｜
for（省略可能式；省略可能式；省略可能式）文；｜
goto 識別子｜識別子：文｜if（式）文｜if（式）文 else 文｜
return；｜return 式；｜switch（式）文｜while（式）文｜
複文｜空文

省略可能式：：＝

式｜

複文：：＝

{宣言リスト　文リスト}｜{宣言子リスト}｜{文リスト}｜
{　}

宣言リスト：：＝

宣言｜宣言リスト　宣言

文リスト：：＝

文｜文リスト　文

空文：：＝

；

5　定　義

ここでのカギカッコ（［、］）は、省略できることを表す。

定義：：＝

関数定義｜データ定義

関数定義：：＝

［記憶クラス指定子］［型指定子］宣言子

（［引数宣言リスト］）複文

引数宣言リスト：：＝

識別子｜引数リスト，識別子

引数宣言：：＝

宣言｜引数宣言　宣言

データ定義：：＝

宣言

6 プリプロセッサ命令

ここでのカギカッコ（［、］）は、省略できることを表す。

プリプロセッサ命令：：＝

　#｜##｜#define 識別子［(［引数リスト］)］［トークン列］｜
　#elif 制限つき定数式｜#else｜#endif｜#if｜制限つき定数式｜
　#ifdef 識別子｜#include 〈ファイル名〉｜
　#include　"ファイル名"｜
　#line 10 進定数［"ファイル名"］｜#undef　識別子

トークン列：：＝

　トークン｜トークン列　トークン

制限つき定数式：：＝

　defined（識別子）｜
　seizeof（式）、（型名）定数式、enum 型定数を除いた整数定数式

●ファイル名の書式

　filename. ext
　　filename ………半角 18 文字、全角 9 文字の長さ
　　ext ………………半角 3 文字、全角 1 文字の長さ

●ファイル名に使用できる文字

　&　#　(　)　@ ^　{　}　!
　英字　数字　下線　半角カナ文字　2 バイト系文字

索引 50音順

記号

ア

イ

エ

オ

カ

マ

メ

モ

ヨ

ロ

シャープ株式会社
本　　社　〒545 大阪市阿倍野区長池町22番22号
電子機器事業本部　〒329-21 栃木県矢板市早川町174番地
液晶映像システム事業部　第2商品企画部
お問い合わせ先　〒162 東京都新宿区市谷八幡町8番地　電話 (03)3260-1161(大代表)
東京支社内　液晶映像システム事業部　第2商品企画部　ソフトウェア担当

③